



First Edition : 2007-2008

Microprocessor & Microcontroller System

A. P. Godse
D. A. Godse



Technical Publications Pune[®]

Microprocessor and Microcontroller System

A. P. Godse

M. S. Software Systems (BITS Pilani)

B.E. Industrial Electronics

Formerly Lecturer in Department of Electronics Engg.
Vishwakarma Institute of Technology
Pune

Mrs. D. A. Godse

B.E. Industrial Electronics, M. E. (Computer)

Assistant Professor in Bharati Vidyapeeth's
Women's College of Engineering
Pune

Visit us at : www.vtubooks.com



Technical Publications Pune®



Microprocessor and Microcontroller System

ISBN 978-81-8431-307-9

All rights reserved with Technical Publications. No part of this book should be reproduced in any form, Electronic, Mechanical, Photocopy or any information storage and retrieval system without prior permission in writing, from Technical Publications, Pune.

Published by :

Technical Publications Pune®

#1, Amit Residency, 412, Shanivar Peth, Pune - 411 030, India.

Printer :

Alert DTPrinters
S.no. 10/3, Sinhagad Road,
Pune - 411 041

Preface

The importance of **Microprocessor and Microcontroller System** is well known in various engineering fields. Overwhelming response to our books on various subjects inspired us to write this book. The book is structured to cover the key aspects of the subject Microprocessor and Microcontroller System.

The book uses plain, lucid language to explain fundamentals of this subject. The book provides logical method of explaining various complicated concepts and stepwise methods to explain the important topics. Each chapter is well supported with necessary illustrations, practical examples and solved problems. All the chapters in the book are arranged in a proper sequence that permits each topic to build upon earlier studies. All care has been taken to make students comfortable in understanding the basic concepts of the subject.

The book not only covers the entire scope of the subject but explains the philosophy of the subject. This makes the understanding of this subject more clear and makes it more interesting. The book will be very useful not only to the students but also to the subject teachers. The students have to omit nothing and possibly have to cover nothing more.

We wish to express our profound thanks to all those who helped in making this book a reality. Much needed moral support and encouragement is provided on numerous occasions by our whole families. We wish to thank the **Publisher** and the entire team of **Technical Publications** who have taken immense pain to get this book in time with quality printing.

Any suggestion for the improvement of the book will be acknowledged and well appreciated.

Authors

A. P. Godse

D. A. Godse

Dedicated to God

Table of Contents

Chapter-1 Introduction to Microprocessor & Microcomputer Systems (1 - 1) to (1 - 26)

| | |
|---|--------|
| 1.1 Introduction to Microprocessor..... | 1 - 1 |
| 1.1.1 Simple Model of Microprocessor | 1 - 3 |
| 1.1.1.1 Counter | 1 - 4 |
| 1.1.1.2 Decoder A | 1 - 4 |
| 1.1.1.3 Register Array | 1 - 4 |
| 1.1.1.4 Common Bus | 1 - 5 |
| 1.1.1.5 Register I | 1 - 6 |
| 1.1.1.6 Decoder B | 1 - 6 |
| 1.1.1.7 Control Unit | 1 - 6 |
| 1.1.1.8 Switch Control Circuit | 1 - 7 |
| 1.1.1.9 ALU | 1 - 7 |
| 1.1.1.10 Operation | 1 - 7 |
| 1.1.2 Terminologies used In Microprocessor..... | 1 - 15 |
| 1.1.3 Different Phases in the Execution Process | 1 - 17 |
| 1.1.3.1 Fetch | 1 - 18 |
| 1.1.3.2 Decode | 1 - 18 |
| 1.1.3.3 Execute | 1 - 18 |
| 1.1.4 Microprocessor Architecture and Its Operation | 1 - 18 |
| 1.2 Microcomputer Systems | 1 - 24 |
| Review Questions | 1 - 25 |

Chapter-2 8085 Microprocessor (2 - 1) to (2 - 50)

| | |
|---|-------|
| 2.1 Introduction | 2 - 1 |
| 2.2 Features of 8085 | 2 - 1 |
| 2.3 Architecture of 8085 | 2 - 2 |
| 2.3.1 Register Structure | 2 - 3 |
| 2.3.2 Arithmetic Logic Unit (ALU) | 2 - 6 |
| 2.3.3 Instruction Decoder | 2 - 6 |
| 2.3.4 Address Buffer | 2 - 7 |

| | |
|---|--------|
| 2.3.5 Address/Data Buffer | 2 - 7 |
| 2.3.6 Incrementer/Decrementer Address Latch | 2 - 7 |
| 2.3.7 Interrupt Control | 2 - 7 |
| 2.3.8 Serial I/O Control | 2 - 7 |
| 2.3.9 Timing and Control Circuitry | 2 - 8 |
| 2.4 Pin Definitions of 8085 | 2 - 8 |
| 2.4.1 Power Supply and Frequency Signals | 2 - 9 |
| 2.4.2 Data Bus and Address Bus | 2 - 9 |
| 2.4.3 Control and Status Signals | 2 - 9 |
| 2.4.4 Interrupt Signals | 2 - 10 |
| 2.4.5 Serial I/O Signals | 2 - 10 |
| 2.4.6 DMA Signal | 2 - 10 |
| 2.4.7 Reset Signals | 2 - 10 |
| 2.5 Clock Circuits | 2 - 11 |
| 2.6 Reset Circuit | 2 - 12 |
| 2.7 8085 Interrupt Structure and Operation | 2 - 13 |
| 2.7.1 Types of Interrupts | 2 - 13 |
| 2.7.2 Overall Interrupt Structure | 2 - 13 |
| 2.7.2.1 Hardware Interrupts in 8085 | 2 - 13 |
| 2.7.2.2 Software Interrupts in 8085 | 2 - 16 |
| 2.7.3 Masking / Unmasking of Interrupts | 2 - 18 |
| 2.7.4 Pending Interrupts | 2 - 19 |
| 2.8 I/O, Memory and System Buses | 2 - 19 |
| 2.8.1 Generation of Address and Data Bus - Demultiplexing $AD_7 - AD_0$ | 2 - 19 |
| 2.8.2 Generation of Control Signals | 2 - 20 |
| 2.8.3 Bus Drivers | 2 - 22 |
| 2.8.4 Typical Configuration | 2 - 24 |
| 2.9 Instruction Cycle | 2 - 25 |
| 2.10 Instruction, Execution, Sequence and Data Flow | 2 - 26 |
| 2.10.1 Representation of Signals | 2 - 27 |
| 2.10.2 Signal Timings | 2 - 29 |
| 2.10.3 Machine Cycles | 2 - 31 |
| 2.10.4 Concept of Wait States | 2 - 48 |
| Review Questions | 2 - 49 |

| | |
|---|---------------|
| 3.1 Introduction | 3 - 1 |
| 3.2 Instruction Classification | 3 - 1 |
| 3.2.1 Data Transfer Operations | 3 - 1 |
| 3.2.2 Arithmetic Operations | 3 - 2 |
| 3.2.3 Logical Operations | 3 - 2 |
| 3.2.4 Branching Operations | 3 - 2 |
| 3.2.5 Stack, Input/Output and Machine Control Operations | 3 - 3 |
| 3.3 Instruction and Data Format | 3 - 3 |
| 3.3.1 Instruction Formats | 3 - 4 |
| 3.3.2 Opcode Formats | 3 - 4 |
| 3.3.3 Data Formats | 3 - 7 |
| 3.4 Instruction Set of 8085 | 3 - 8 |
| 3.4.1 Data Transfer Group | 3 - 8 |
| 3.4.2 Arithmetic Group | 3 - 11 |
| 3.4.3 Logic Group | 3 - 18 |
| 3.4.4 Rotate Group | 3 - 23 |
| 3.4.5 Stack Operations | 3 - 25 |
| 3.4.6 Branch Group | 3 - 29 |
| 3.4.7 Input/Output | 3 - 33 |
| 3.4.8 Machine Control Group | 3 - 34 |
| 3.5 Addressing Modes | 3 - 35 |
| 3.6 Instruction Set Summary | 3 - 37 |
| 3.7 Assembly Language Programming | 3 - 41 |
| 3.7.1 Steps Involved in Programming | 3 - 42 |
| 3.7.2 Flowchart | 3 - 42 |
| 3.7.3 Assembly Language Program | 3 - 43 |
| 3.7.4 Assembly Language Program to Machine Language Program | 3 - 44 |
| 3.7.5 Storing Hex Code in the Memory | 3 - 45 |
| 3.7.6 Executing the Program | 3 - 45 |
| 3.8 Programming Examples | 3 - 46 |
| 3.9 Instruction Comparisons | 3 - 62 |
| Review Questions | 3 - 65 |

| | |
|---|---------------|
| 4.1 Looping, Counting and Indexing | 4 - 1 |
| 4.2 Timers | 4 - 33 |
| 4.2.1 Timer Delay using NOP Instruction | 4 - 34 |
| 4.2.2 Timer Delay using Counters | 4 - 34 |
| 4.2.3 Timer Delay using Nested Loops | 4 - 36 |
| 4.3 Code Conversion | 4 - 43 |
| 4.3.1 BCD to Binary Conversion | 4 - 43 |
| 4.3.2 Binary to BCD Conversion | 4 - 44 |
| 4.3.3 BCD to Seven Segment Conversion | 4 - 47 |
| 4.3.4 Binary to ASCII Code Conversion | 4 - 48 |
| 4.3.5 ASCII Code to Binary Conversion | 4 - 50 |
| 4.4 BCD Arithmetic | 4 - 50 |
| 4.4.1 BCD Addition | 4 - 50 |
| 4.4.2 BCD Subtraction | 4 - 54 |
| Review Questions | 4 - 55 |

| | |
|--|---------------|
| 5.1 Concept of Stack and Subroutines | 5 - 1 |
| 5.1.1 Stack | 5 - 1 |
| 5.1.1.1 Stack Related Instructions | 5 - 2 |
| 5.1.1.2 Detail Operation and the use of Stack Related Instructions | 5 - 3 |
| 5.1.2 Subroutines | 5 - 11 |
| 5.2 Parameters Passing Techniques | 5 - 13 |
| 5.2.1 Passing Parameters using Registers | 5 - 13 |
| 5.2.2 Passing Parameters using Memory | 5 - 14 |
| 5.2.3 Passing Parameters using Pointers | 5 - 14 |
| 5.2.4 Passing Parameters using Stack | 5 - 15 |
| 5.3 Subroutine Documentation | 5 - 17 |
| 5.4 Advanced Subroutine Concepts | 5 - 18 |
| 5.4.1 Re-entrant Subroutine | 5 - 18 |
| 5.4.2 Recursive Subroutine | 5 - 19 |
| Review Questions | 5 - 21 |

| | |
|--|---------------|
| 6.1 Introduction | 6 - 1 |
| 6.1.1 Input / Output System | 6 - 1 |
| 6.1.2 Requirements of I/O System | 6 - 2 |
| 6.1.3 I/O Ports | 6 - 4 |
| 6.2 I/O Data Transfer Techniques | 6 - 5 |
| 6.2.1 Programmed I/O | 6 - 6 |
| 6.2.2 Interrupt Driven I/O | 6 - 8 |
| 6.2.3 Comparison between Programmed I/O and Interrupt Driven I/O | 6 - 9 |
| 6.2.4 DMA Transfer | 6 - 9 |
| 6.2.4.1 Hardware Controlled Data Transfer | 6 - 10 |
| 6.2.4.2 DMA Idle Cycle | 6 - 11 |
| 6.2.4.3 DMA Active Cycle | 6 - 11 |
| 6.2.5 Data Transfer Modes | 6 - 12 |
| 6.3 I/O Interfacing Techniques in 8085 | 6 - 15 |
| 6.3.1 I/O Mapped I/O | 6 - 15 |
| 6.3.2 I/O Device Selection | 6 - 16 |
| 6.3.3 Interfacing Input and Output Devices with Examples | 6 - 17 |
| 6.3.4 Memory Mapped I/O | 6 - 21 |
| 6.3.5 Comparison between I/O Mapped I/O and Memory Mapped I/O | 6 - 23 |
| 6.4 Memory Interfacing | 6 - 24 |
| 6.4.1 Basic Concepts in Memory Interfacing | 6 - 24 |
| 6.4.2 Interfacing Examples | 6 - 28 |
| Review Questions | 6 - 37 |

| | |
|--|--------------|
| 7.1 Introduction | 7 - 1 |
| 7.2 Features of 8051 | 7 - 3 |
| 7.3 8051 Microcontroller Hardware | 7 - 3 |
| 7.3.1 Pin-out of 8051 | 7 - 5 |
| 7.3.2 Central Processing Unit (CPU) | 7 - 7 |
| 7.3.3 Internal RAM | 7 - 7 |
| 7.3.4 Internal ROM | 7 - 9 |
| 7.3.5 Input/Output Ports | 7 - 9 |
| 7.3.6 Register Set of 8051 | 7 - 10 |

| | |
|--|---------------|
| 7.3.6.1 Register A (Accumulator) | 7 - 10 |
| 7.3.6.2 Register B | 7 - 11 |
| 7.3.6.3 Program Status Word (Flag Register) | 7 - 11 |
| 7.3.6.4 Stack and Stack Pointer | 7 - 11 |
| 7.3.6.5 Data Pointer (DPTR) | 7 - 12 |
| 7.3.6.6 Program Counter | 7 - 12 |
| 7.3.6.7 Special Function Registers | 7 - 13 |
| 7.4 Memory Organization in 8051 | 7 - 15 |
| 7.5 Input/Output Pins, Ports and Circuits | 7 - 16 |
| 7.6 Timers and Counters | 7 - 19 |
| 7.6.1 Timer/Counter Control Logic | 7 - 19 |
| 7.6.2 Timer 0 and Timer 1 | 7 - 20 |
| 7.7 Serial Port | 7 - 24 |
| 7.7.1 Operating Modes for Serial Port | 7 - 25 |
| 7.7.2 Serial Port Control Register | 7 - 26 |
| 7.7.3 Generating Baud Rates | 7 - 26 |
| 7.8 Interrupt Structure | 7 - 28 |
| 7.8.1 Priority Level Structure | 7 - 29 |
| 7.8.2 External Interrupts | 7 - 31 |
| 7.8.3 Single-Step Operation | 7 - 31 |
| 7.9 Clock and Oscillator | 7 - 31 |
| 7.10 Power Saving Options | 7 - 33 |
| 7.10.1 Idle Mode | 7 - 34 |
| 7.10.2 Power Down Mode | 7 - 34 |
| 7.11 Multiprocessor Communication in MCS-51 | 7 - 37 |
| Review Questions | 7 - 37 |

Chapter-8 Instruction Set and Programming of 8051 (8 - 1) to (8 - 88)

| | |
|--|---------------|
| 8.1 Addressing Modes | 8 - 1 |
| 8.2 Data Moving (Data Transfer) Instructions | 8 - 4 |
| 8.2.1 Instructions to Access External Data Memory | 8 - 8 |
| 8.2.2 Instructions to Access External ROM / Program Memory | 8 - 11 |
| 8.2.3 PUSH and POP Instructions | 8 - 13 |
| 8.2.3.1 PUSH | 8 - 13 |
| 8.2.3.2 POP | 8 - 14 |
| 8.2.4 Data Exchange Instructions | 8 - 16 |
| 8.3 Logical Instructions | 8 - 18 |

| | |
|---|--------|
| 8.3.1 Byte Level Logical Operations | 8 - 18 |
| 8.3.2 Bit Level Logical Operations | 8 - 26 |
| 8.3.3 Rotate and Swap Operations | 8 - 31 |
| 8.4 Arithmetic Instructions..... | 8 - 34 |
| 8.4.1 Flags | 8 - 34 |
| 8.4.2 Incrementing and Decrementing | 8 - 35 |
| 8.4.3 Addition | 8 - 38 |
| 8.4.4 Subtraction | 8 - 40 |
| 8.4.5 Multiplication and Division | 8 - 42 |
| 8.4.6 Decimal Arithmetic | 8 - 44 |
| 8.5 Jump and CALL Instructions..... | 8 - 46 |
| 8.5.1 Jump and Call Program Range | 8 - 47 |
| 8.5.2 Jump | 8 - 47 |
| 8.5.3 CALL and Subroutines | 8 - 58 |
| 8.6 Programming Examples..... | 8 - 61 |
| Review Questions | 8 - 86 |

Chapter-9 Counter/Timer Programming in 8051 (9 - 1) to (9 - 14)

| | |
|-------------------------------------|--------|
| 9.1 8051 Timers | 9 - 1 |
| 9.1.1 Timer Registers | 9 - 1 |
| 9.1.2 Programming 8051 Timers | 9 - 3 |
| 9.1.2.1 Mode 0 | 9 - 3 |
| 9.1.2.2 Mode 1 | 9 - 4 |
| 9.1.2.3 Mode 2 | 9 - 5 |
| 9.1.2.4 Mode 3 | 9 - 5 |
| 9.1.3 Loading Timer Registers | 9 - 8 |
| 9.2 Programming Counters | 9 - 9 |
| Review Questions | 9 - 13 |

Chapter-10 8051 Serial Communication (10 - 1) to (10 - 10)

| | |
|--|--------|
| 10.1 8051 Connections to RS 232C | 10 - 1 |
| 10.2 8051 Serial Communication Programming | 10 - 2 |
| 10.2.1 Operating Modes for Serial Port | 10 - 2 |
| 10.2.2 SBUF | 10 - 3 |
| 10.2.3 Serial Port Control Register | 10 - 4 |
| 10.2.4 Generating Baud Rates | 10 - 5 |

| | |
|---|---------|
| 10.2.5 Programming 8051 for Serial Data Transfer | 10 - 6 |
| 10.2.5.1 Importance of the TI Flag Bit | 10 - 8 |
| 10.2.6 Programming 8051 for Receiving Serial Data | 10 - 8 |
| 10.2.6.1 Importance of the RI Flag Bit | 10 - 9 |
| 10.2.7 Doubling the Baud Rate in the 8051 | 10 - 9 |
| Review Questions | 10 - 10 |

Chapter-11 Interrupt Programming (11 - 1) to (11 - 10)

| | |
|--|--------|
| 11.1 8051 Interrupts | 11 - 1 |
| 11.2 Interrupt Vector Table | 11 - 2 |
| 11.3 Enabling and Disabling an Interrupt | 11 - 2 |
| 11.4 Timer Interrupts and Programming | 11 - 3 |
| 11.5 Programming External Hardware Interrupts | 11 - 4 |
| 11.6 Serial Communication Interrupts and Programming | 11 - 5 |
| 11.7 Interrupt Priority in the 8051 | 11 - 7 |
| 11.7.1 Nested Interrupts | 11 - 8 |
| 11.7.2 Software Triggering of Interrupt | 11 - 8 |
| Review Questions | 11 - 8 |

Chapter-12 Programmable Peripheral Interface-8255 (12 - 1) to (12 - 26)

| | |
|---|---------|
| 12.1 Features of 8255A | 12 - 1 |
| 12.2 Pin Diagram | 12 - 2 |
| 12.3 Block Diagram | 12 - 4 |
| 12.3.1 Data Bus Buffer | 12 - 5 |
| 12.3.2 Control Logic | 12 - 5 |
| 12.3.3 Group A and Group B Controls | 12 - 5 |
| 12.4 Operation Modes | 12 - 5 |
| 12.4.1 Bit Set-Reset (BSR) Mode | 12 - 5 |
| 12.4.2 I/O Modes | 12 - 6 |
| 12.5 Control Word Formats | 12 - 7 |
| 12.5.1 For Bit Set/Reset Mode | 12 - 7 |
| 12.5.2 For I/O Mode | 12 - 8 |
| 12.6 8255 Programming and Operation | 12 - 11 |
| 12.6.1 Programming in Mode 0 | 12 - 11 |
| 12.6.2 Programming in Mode 1 (Input / Output with Handshake) | 12 - 13 |
| 12.6.3 Programming in Mode 2 (Strobes Bi-directional Bus I/O) | 12 - 19 |
| 12.7 Interfacing 8255 in I/O Mapped I/O | 12 - 22 |

| | |
|--|---------|
| 12.8 Interfacing 8255 in Memory Mapped I/O | 12 - 23 |
| Review Questions | 12 - 26 |

Chapter-13 Interfacing to External World (13 - 1) to (13 - 66)

| | |
|--|---------|
| 13.1 Introduction | 13 - 1 |
| 13.2 External RAM and ROM | 13 - 1 |
| 13.2.1 Program Memory | 13 - 1 |
| 13.2.2 Data Memory | 13 - 3 |
| 13.2.3 Important Points to Remember in Accessing External Memory | 13 - 6 |
| 13.2.4 Memory Interfacing | 13 - 6 |
| 13.2.5 Interfacing Example | 13 - 10 |
| 13.3 8051 I/O Expansion using 8255 | 13 - 12 |
| 13.4 Interfacing Keyboard | 13 - 13 |
| 13.4.1 Key Debounce using Hardware | 13 - 14 |
| 13.4.2 Key Debouncing using Software | 13 - 15 |
| 13.4.3 Simple Keyboard Interface | 13 - 15 |
| 13.4.4 Matrix Keyboard Interface | 13 - 17 |
| 13.5 Interfacing Display | 13 - 21 |
| 13.5.1 LED Displays | 13 - 21 |
| 13.5.2 Interfacing LED Displays | 13 - 23 |
| 13.6 Interfacing LCD Display | 13 - 27 |
| 13.7 Interfacing DAC to 8051 | 13 - 34 |
| 13.7.1 IC DAC 1408 | 13 - 34 |
| 13.7.1.1 Important Electrical Characteristics for IC 1408 | 13 - 38 |
| 13.7.2 Interfacing DAC 1408 / DAC 0808 with 8051 | 13 - 38 |
| 13.8 Interfacing ADC to 8051 | 13 - 42 |
| 13.8.1 ADC 0804 Family | 13 - 42 |
| 13.8.2 ADC 0808/0809 Family | 13 - 44 |
| 13.8.3 Interfacing of ADC 0803/0804/0805 with 8051 | 13 - 49 |
| 13.9 Stepper Motor Interface | 13 - 50 |
| 13.10 Typical MCS-51 Based System | 13 - 55 |
| 13.11 Interfacing Examples | 13 - 56 |
| Review Questions | 13 - 65 |

| | |
|--|----------------|
| 14.1 Introduction | 14 - 1 |
| 14.2 Classification of Serial Data Communication | 14 - 2 |
| 14.2.1 Simplex | 14 - 2 |
| 14.2.2 Half Duplex | 14 - 2 |
| 14.2.3 Full Duplex | 14 - 2 |
| 14.3 Basic Serial Communication Data Formats | 14 - 2 |
| 14.3.1 Asynchronous | 14 - 2 |
| 14.3.2 Synchronous | 14 - 3 |
| 14.4 RS-232C | 14 - 4 |
| 14.4.1 DB-25 P Connector | 14 - 4 |
| 14.4.2 DB-9P Connector | 14 - 6 |
| 14.4.3 Data Transmission and Reception using RS-232C | 14 - 7 |
| 14.4.4 8051 Connection to RS-232C | 14 - 8 |
| 14.5 RS-423-A | 14 - 9 |
| 14.6 RS-422-A | 14 - 11 |
| 14.7 RS-485 | 14 - 12 |
| 14.8 Comparison between EIA Standards | 14 - 13 |
| 14.9 I²C | 14 - 14 |
| 14.9.1 Features of I ² C Bus | 14 - 14 |
| 14.9.2 I ² C Bus Terminology | 14 - 15 |
| 14.9.3 I ² C BUS Configuration | 14 - 15 |
| 14.9.4 I ² C Signals | 14 - 16 |
| 14.9.5 Initiating and Terminating Data Transfer | 14 - 16 |
| 14.9.6 Transferring Data | 14 - 17 |
| 14.9.6.1 Byte Format | 14 - 17 |
| 14.9.6.2 Acknowledge | 14 - 18 |
| 14.9.7 Addressing I ² C Devices | 14 - 18 |
| 14.9.8 Complete Data Transfer | 14 - 19 |
| 14.9.9 Arbitration | 14 - 21 |
| 14.9.10 Extensions to the Standard-Mode I ² C Bus Specification | 14 - 22 |
| 14.9.10.1 Fast-mode | 14 - 22 |
| 14.9.10.2 Hs-mode | 14 - 23 |
| 14.9.11 Advantages of I ² C | 14 - 23 |
| 14.9.12 Disadvantage of I ² C | 14 - 24 |

| | |
|---|----------------|
| 14.9.13 Applications of IC | 14 - 24 |
| 14.10 Modbus Protocol | 14 - 24 |
| 14.10.1 Transactions on Modbus Networks | 14 - 24 |
| 14.10.2 The Query-Response Cycle | 14 - 25 |
| 14.10.3 Transmission Modes | 14 - 26 |
| 14.10.3.1 ASCII Mode | 14 - 26 |
| 14.10.3.2 RTU Mode | 14 - 27 |
| 14.10.4 Modbus Message Framing | 14 - 27 |
| 14.10.4.1 ASCII Framing | 14 - 27 |
| 14.10.4.2 RTU Framing | 14 - 28 |
| 14.10.4.3 More About Address Field | 14 - 28 |
| 14.10.4.4 More About Function Field | 14 - 29 |
| 14.10.4.5 More About Data Field | 14 - 29 |
| 14.10.4.6 More About Error Checking Field | 14 - 30 |
| 14.10.5 Error Checking Methods | 14 - 30 |
| 14.10.5.1 Parity Checking | 14 - 30 |
| 14.10.5.2 Frame Checking | 14 - 31 |
| 14.11 IEEE 488 Standard | 14 - 32 |
| Review Questions | 14 - 35 |

Chapter-15 Interfacing to EEPROM and RTC DS1307 (15 - 1) to (15 - 18)

| | |
|--|----------------|
| 15.1 Serial EEPROMs | 15 - 1 |
| 15.2 Types of Serial EEPROM Chips | 15 - 1 |
| 15.3 EEPROM : 93C46/56/66 | 15 - 1 |
| 15.4 EEPROM : 24C16 | 15 - 6 |
| 15.5 EEPROM : 24C32/64 | 15 - 8 |
| 15.6 RTC DS1307 | 15 - 11 |
| 15.6.1 Features of DS1307 | 15 - 11 |
| 15.6.2 DS1307 Operation | 15 - 11 |
| 15.6.3 DS1307 Block Diagram | 15 - 12 |
| 15.6.4 Signal Descriptions | 15 - 12 |
| 15.6.5 RTC and RAM Address Map | 15 - 13 |
| 15.6.6 Clock and Calender | 15 - 13 |
| 15.6.7 Control Register | 15 - 14 |
| 15.6.8 2-Wire Serial Data Bus | 15 - 15 |
| 15.6.9 Operating Modes of DS1307 | 15 - 17 |

| | |
|------------------------|---------|
| Review Questions | 15 - 18 |
|------------------------|---------|

Chapter-16 Conceptual Study of Derivatives of Microcontroller (16 - 1) to (16 - 14)

| | |
|--|---------|
| 16.1 Introduction | 16 - 1 |
| 16.2 One-Time-Programmable (OTP) Feature | 16 - 1 |
| 16.3 Features of 89C51 Microcontroller | 16 - 2 |
| 16.4 Microcontroller – 89C51 RD | 16 - 3 |
| 16.4.1 Features and 89C51 RD | 16 - 3 |
| 16.4.2 Block Diagram | 16 - 4 |
| 16.4.3 Stop Clock Mode | 16 - 4 |
| 16.4.4 Enhanced UART | 16 - 4 |
| 16.4.5 Automatic Address Recognition | 16 - 7 |
| 16.4.6 Dual DPTR | 16 - 8 |
| 16.4.7 Programmable Counter Array (PCA) | 16 - 8 |
| 16.5 Microcontroller Supervisory Control | 16 - 12 |
| Review Questions | 16 - 13 |

Chapter-17 AVR Microcontrollers (17 - 1) to (17 - 20)

| | |
|--|---------|
| 17.1 Introduction | 17 - 1 |
| 17.2 Features of AT90S2313 Microcontroller | 17 - 2 |
| 17.3 Architecture of AT90S2313 | 17 - 3 |
| 17.4 Memory Map of AVR AT90S2313 | 17 - 7 |
| 17.5 Memory Access and Instruction Execution | 17 - 8 |
| 17.6 I/O Memory | 17 - 9 |
| 17.7 Reset and Interrupt Handling | 17 - 10 |
| 17.8 Registers in AT90S2313 | 17 - 11 |
| 17.8.1 General Interrupt Mask Register - GIMSK | 17 - 11 |
| 17.8.2 General Interrupt FLAG Register – GIFR | 17 - 12 |
| 17.8.3 Timer/Counter Interrupt Mask Register – TIMSK | 17 - 12 |
| 17.8.4 Timer/Counter Interrupt FLAG Register – TIFR | 17 - 13 |
| 17.8.5 MCU Control Register – MCUCR | 17 - 14 |
| 17.8.6 Timer/Counters Control Registers | 17 - 14 |
| 17.8.6.1 Timer/Counter1 Control Register A – TCCR1A | 17 - 15 |
| 17.8.6.2 Timer/Counter1 Control Register B – TCCR1B | 17 - 16 |
| 17.8.6.3 Timer/Counter1 – TCNT1H and TCNT1L | 17 - 17 |

| | |
|---|---------|
| 17.8.6.4 Timer/Counter1 Output Compare Register A – OCR1AH and OCR1AL . . . | 17 - 17 |
| 17.8.6.5 Timer/Counter1 Input Capture Register – ICR1H and ICR1L | 17 - 18 |
| 17.8.7 Watchdog Timer Control Register – WDTCR | 17 - 18 |
| 17.8.8 EEPROM Read/Write Access Registers | 17 - 19 |
| 17.8.8.1 EEPROM Address Register – EEAR | 17 - 19 |
| 17.8.8.2 EEPROM Data Register – EEDR | 17 - 19 |
| 17.8.8.3 EEPROM Control Register – EECR | 17 - 20 |
| Review Questions | 17 - 20 |

Chapter-18 Introduction to PIC Microcontroller (18 - 1) to (18 - 22)

| | |
|---|---------|
| 18.1 Introduction | 18 - 1 |
| 18.2 Features of PIC Microcontrollers | 18 - 1 |
| 18.3 PIC16CXX | 18 - 5 |
| 18.3.1 Features of 16C6X Microcontroller | 18 - 5 |
| 18.3.1.1 Core Features | 18 - 5 |
| 18.3.1.2 Peripheral Features | 18 - 6 |
| 18.3.2 Block Diagram | 18 - 7 |
| 18.3.2.1 Harvard Architecture | 18 - 7 |
| 18.3.2.2 Address and Data Bus | 18 - 7 |
| 18.3.2.3 ALU | 18 - 8 |
| 18.3.2.4 Registers | 18 - 8 |
| 18.3.2.5 Clocking Scheme/Instruction Cycle. | 18 - 12 |
| 18.3.3 Pin Diagram of PIC16C61 | 18 - 13 |
| 18.3.4 Pin Diagram of PIC16C71 | 18 - 15 |
| 18.3.5 PIC Memory Organization | 18 - 15 |
| 18.3.5.1 Program Memory. | 18 - 15 |
| 18.3.5.2 Data Memory | 18 - 16 |
| 18.3.6 Watchdog Timer | 18 - 20 |
| Review Questions | 18 - 21 |

8085 Programs

| | |
|--|--------|
| Lab Experiment 1 : Store 8-bit data in memory. | 3 - 46 |
| Lab Experiment 2 : Exchange the contents of memory locations. | 3 - 46 |
| Lab Experiment 3 : Add two 8-bit numbers. | 3 - 47 |
| Lab Experiment 4 : Subtract two 8-bit numbers. | 3 - 47 |
| Lab Experiment 5 : Add two 16-bit numbers. | 3 - 48 |
| Lab Experiment 6 : Subtract two 16-bit numbers. | 3 - 49 |
| Lab Experiment 7 : Check results after execution of INR B, INR C and INX B instructions. | 3 - 50 |
| Lab Experiment 8 : Check results after execution of DCR C, DCR B and DCX B instructions. | 3 - 51 |
| Lab Experiment 9 : Find the 1's complement of a number. | 3 - 52 |
| Lab Experiment 10 : Find the 2's complement of a number. | 3 - 52 |
| Lab Experiment 11 : Pack the two unpacked BCD numbers. | 3 - 53 |
| Lab Experiment 12 : Unpack the BCD number. | 3 - 53 |
| Lab Experiment 13 : Sample subroutine program. | 3 - 54 |
| Lab Experiment 14 : Add contents of two memory locations. | 3 - 55 |
| Lab Experiment 15 : Right shift data within the 8-bit register. | 3 - 56 |
| Lab Experiment 16 : Right shift data with in 16-bit register. | 3 - 57 |
| Lab Experiment 17 : Left shift 16-bit data within 16-bit register. | 3 - 58 |
| Lab Experiment 18 : Alter the contents of flag register. | 3 - 58 |
| Lab Experiment 19 : Find 2's complement of 16-bit number. | 3 - 59 |
| Lab Experiment 20 : Simulation of CALL and RET instructions. | 3 - 59 |
| Lab Experiment 21 : Find the factorial of a number. | 3 - 60 |
| Lab Experiment 22 : Calculate the sum of series of numbers. | 4 - 3 |
| Lab Experiment 23 : Data transfer from memory block B1 to memory block B2. | 4 - 5 |
| Lab Experiment 24 : Multiply two 8-bit numbers. | 4 - 6 |
| Lab Experiment 25 : Divide 16-bit number by 8-bit number. | 4 - 7 |
| Lab Experiment 26 : Find the negative numbers in a block of data. | 4 - 8 |
| Lab Experiment 27 : Find the largest of given numbers. | 4 - 10 |
| Lab Experiment 28 : Count number of one's in a number. | 4 - 11 |
| Lab Experiment 29 : Arrange numbers in the ascending order. | 4 - 12 |
| Lab Experiment 30 : Calculate the sum of series of even numbers. | 4 - 14 |
| Lab Experiment 31 : Calculate the sum of series of odd numbers. | 4 - 15 |
| Lab Experiment 32 : Find the square of given number. | 4 - 17 |
| Lab Experiment 33 : Search a byte in a given number. | 4 - 18 |
| Lab Experiment 34 : Add two decimal numbers. | 4 - 19 |
| Lab Experiment 35 : Add each element of array with the elements of another array. | 4 - 20 |
| Lab Experiment 36 : Separate even numbers from given numbers. | 4 - 21 |

| | |
|---|---------|
| Lab Experiment 37 : Transfer contents to overlapping memory blocks. | 4 - 21 |
| Lab Experiment 38 : Inserting string in a given array of characters. | 4 - 23 |
| Lab Experiment 39 : Deleting string in a given array of characters. | 4 - 23 |
| Lab Experiment 40 : Add parity bit to 7-bit ASCII characters. | 4 - 24 |
| Lab Experiment 41 : Find the number of negative, zero and positive numbers. | 4 - 25 |
| Lab Experiment 42 : Multiply two eight bit numbers with shift and add method.. | 4 - 26 |
| Lab Experiment 43 : Divide 16-bit number with 8-bit number using shifting technique. | 4 - 28 |
| Lab Experiment 44 : Simulate DAA instruction.. | 4 - 30 |
| Lab Experiment 45 : Program to test RAM. | 4 - 32 |
| Lab Experiment 46 : Write an assembly language program to generate fibonacci number. | 4 - 33 |
| Lab Experiment 47 : Generate a delay of 0.4 seconds. | 4 - 36 |
| Lab Experiment 48 : Generate and display binary up counter. | 4 - 37 |
| Lab Experiment 49 : Generate and display BCD up counter with frequency 1 Hz.. | 4 - 38 |
| Lab Experiment 50 : Generate and display BCD down counter with frequency 1 Hz | 4 - 39 |
| Lab Experiment 51 : Generate and display the contents of decimal counter. | 4 - 41 |
| Lab Experiment 52 : Identify the error and correct the given delay routine. | 4 - 42 |
| Lab Experiment 53 : 2-Digit BCD to binary conversion. | 4 - 43 |
| Lab Experiment 54 : Binary to BCD conversion. | 4 - 45 |
| Lab Experiment 55 : Find the 7-segment codes for given numbers. | 4 - 47 |
| Lab Experiment 56 : Find the ASCII character. | 4 - 48 |
| Lab Experiment 57 : Add two 2-digit BCD numbers. | 4 - 52 |
| Lab Experiment 58 : Add two 4-digit BCD numbers. | 4 - 52 |
| Lab Experiment 59 : Subtraction of two BCD numbers. | 4 - 54 |
| Lab Experiment 60 : Multiply two 2-digit BCD numbers. | 4 - 54 |
| Lab Experiment 61 : Solve given equation. | 5 - 22 |
| Lab Experiment 62 : Blink port C bit 0 of 8255. | 12 - 10 |
| Lab Experiment 63 : Hardware and software for flashing LEDs. | 12 - 24 |

Introduction to Microprocessor and Microcomputer Systems

1.1 Introduction to Microprocessor

This chapter, the first of this text, is written primarily to introduce to the student the microprocessor based system and the role of microprocessor in it. It also explains various components required to build a microprocessor system and the working of generalized microprocessor unit.

What is a Microprocessor Based System? Don't worry, we shall get fair idea about this on completion of this chapter.

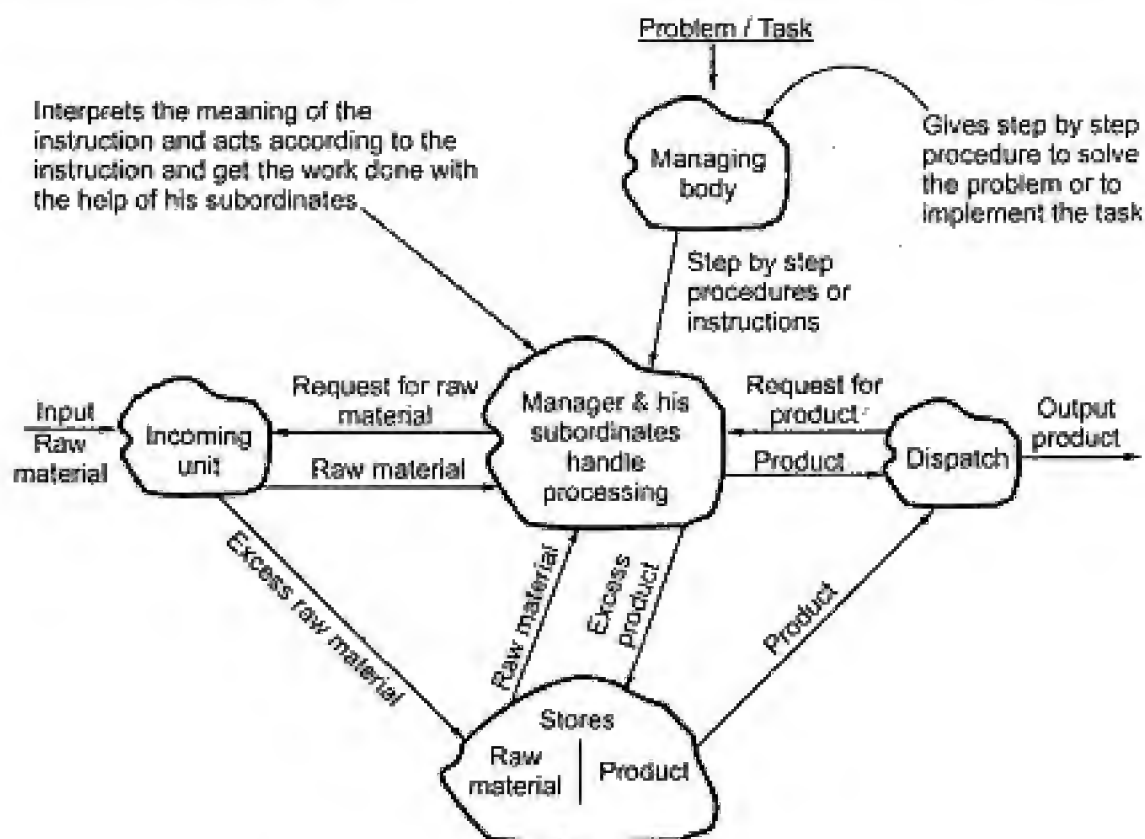


Fig. 1.1 Working of small organization

As a part of introduction to the microprocessor based system, I would like to share philosophy behind this microprocessor based systems. Let us consider a small organization. It consists of managing body, manager and his subordinates, incoming unit and dispatch unit. (To maintain the simplicity and just to explain analogy I have considered small organization as shown in the Fig. 1.1) (See Fig. 1.1 on previous page).

As shown in the Fig. 1.1, managing body gives step by step procedure to solve the problem or to implement the task. Manager interprets the step by step procedures and instructions given by the managing body and executes those with the help of his subordinates. The manager and his subordinates who belong to processing unit can request for raw material to stores. If raw material is not available in stores, processing unit can request for raw material to incoming unit. After processing the raw material, processing unit can either dispatch product or keep it in stores.

Looking at this environment somebody came up with ideas.

- Can we design such an organization without any involvement human beings ?
- Can we replace human being with some electronic devices which can mimic the operations executed by human beings ?.

In this way the idea of microprocessor based system was born and design process was started. Over the period of design process, designers first came up with an integrated circuit which has an ability to mimic the processes executed by the processing unit. They replaced raw material by data and instructions by code with 1s and 0s. The integrated circuit accepts data as an input and process it according to instruction code to generate an information. As an example, we can say, marks of students in various subjects is a data and result prepared on the basis of these marks is an information.

To get the data as an input and to give information to the users, the designer team came up with the input and output devices which is an another integrated circuit. Therefore, with the use of input device processing unit can get the input and after processing the input, processor will give the information to the user with the help of output device. The integrated circuit was also designed which has an ability to store the data, instruction code and information. The processing unit reads instruction code from this circuit and act accordingly. Some meaningful name was given to each integrated circuit as given below.

For Processing Unit Integrated Circuit :

As it is a heart of organization and has an ability to process input as per the instructions of managing body, it was named 'Central Processing Unit' (CPU) or simply 'Processor'.

For Incoming Unit Integrated Circuit :

As it gives input data to the processor it was named **Input device**.

For Dispatch Unit Integrated Circuit :

As it gives information to the user, it was named **Output device**.

For Storage Unit Integrated Circuit :

As it stores data instruction code and information, it was called **Memory**.

The system with central processing unit, input device, output device and memory can be visualized as shown in the Fig. 1.2. The system designed here is very small in physical size as compared to the organization with a human being. The processor is an integrated circuit in a very small size and hence more appropriate name for it was suggested as a '**Microprocessor**'. The system as a whole was then referred to as '**Microprocessor Based System**.'

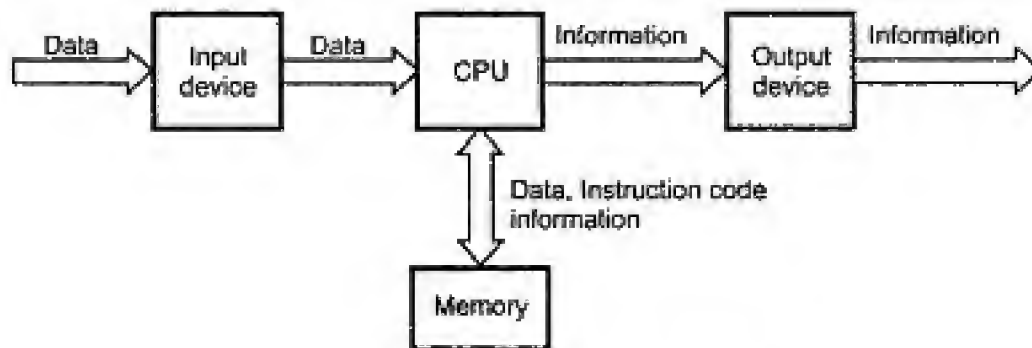


Fig. 1.2 Simple microprocessor based system

1.1.1 Simple Model of Microprocessor

In the last section we have seen the basic blocks of the microprocessor based system. Let us discuss each block with some details. This section explains the working of microprocessor block with help of simple model.

Simple model is a sequential digital circuit as shown in the Fig. 1.3. It consists of 4 bit up-counter, decoders, registers, switch control circuit, control unit and arithmetic logic unit (ALU).

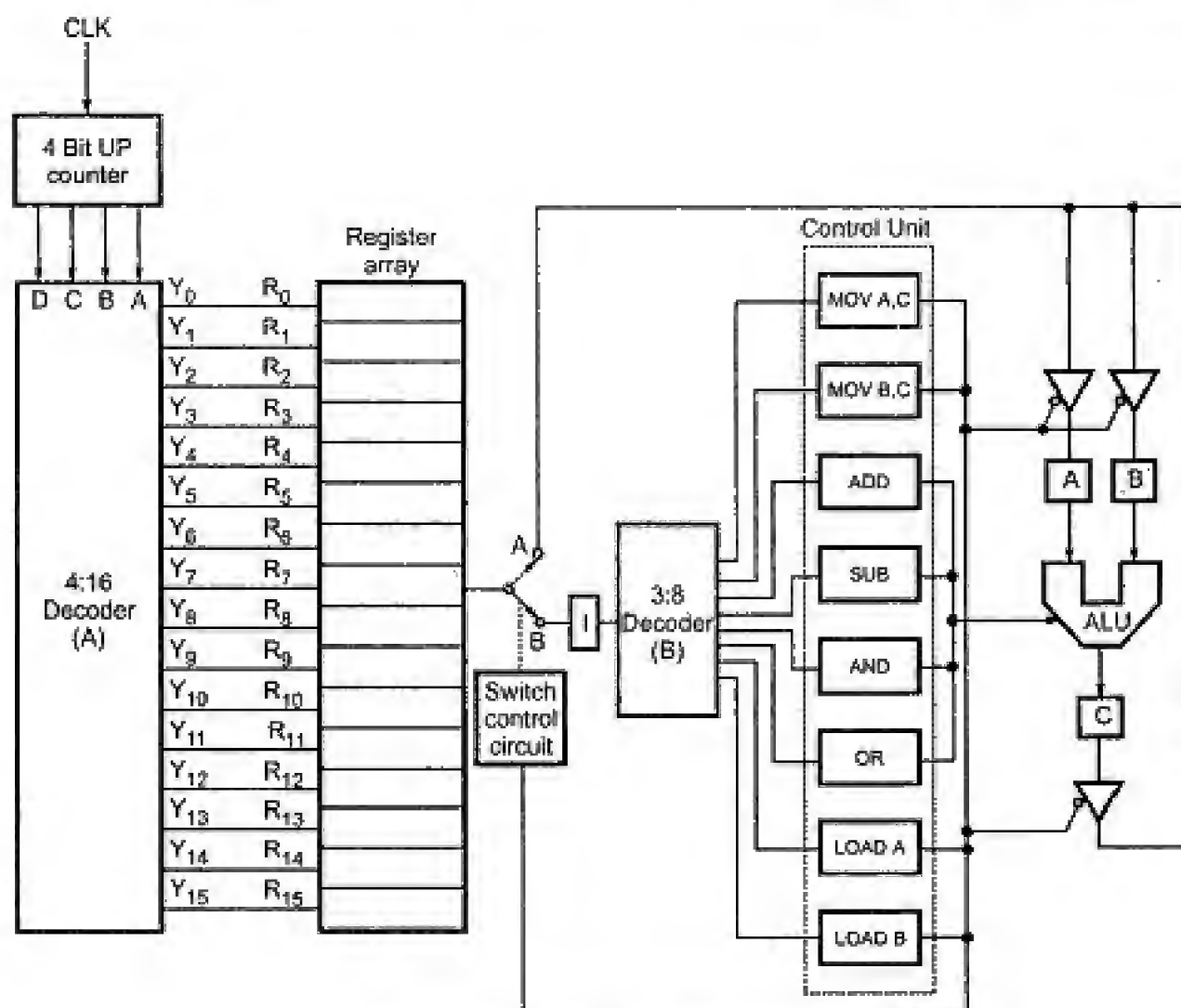


Fig. 1.3 Sequential digital circuit

1.1.1.1 Counter

It is a 4 bit up-counter. It counts from 0 to 15 and it always starts from zero. The counter is driven by the clock signal and the output of counter is given to the decoder A.

1.1.1.2 Decoder A

It is a 4:16 decoder. The four bit output of counter is used as an input for the decoder A. It activates its output depending on the status of inputs. The output of the decoder A is used to select one of the registers within the register array.

1.1.1.3 Register Array

Register array consists of sixteen 3 bit registers. Each register can store 3 bit word. The outputs of all registers are connected to the common bus through the Output Enable (OE) switch as shown in Fig. 1.4.

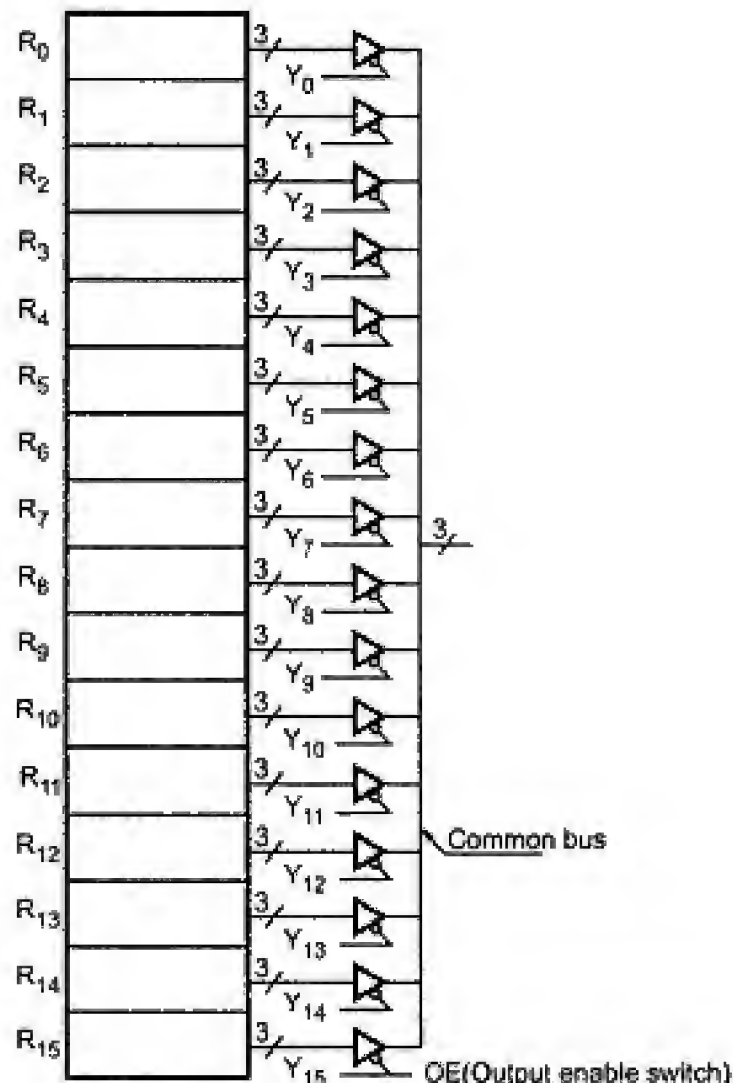


Fig. 1.4 Register array with tri-state switches

The output of the decoder A is used to activate output enable signals for sixteen registers (R_0 - R_{15}). At a time only one output of decoder is activated according to the input signal, so data from only one register is available on the common bus.

1.1.1.4 Common Bus

The three bit common bus is connected to register A, register B, and to the register I through the switch. If the switch is at position A then common bus is connected to the A

or B register; otherwise it is connected to the register I. The switch position is controlled by the switch control circuit.

1.1.1.5 Register I

It is used to store data from the common bus.

1.1.1.6 Decoder B

It is a 3:8 decoder. The data from the Register I is used as an input for the decoder. If switch is in position B, this data is decoded and used to select appropriate control circuit from the control unit as shown in Table 1.1.

| D ₂ D ₁ D ₀ | Selected Control Circuit |
|--|--------------------------|
| 0 0 0 | MOV A, C |
| 0 0 1 | MOV B, C |
| 0 1 0 | ADD |
| 0 1 1 | SUB |
| 1 0 0 | AND |
| 1 0 1 | OR |
| 1 1 0 | LOAD A |
| 1 1 1 | LOAD B |

Table 1.1

1.1.1.7 Control Unit

It consists of eight control circuits: MOV A, C, MOV B, C, ADD, SUB, AND, OR, LOAD A and LOAD B. These control circuits are used to generate signals which select the operation of ALU and activate input enable signal of register A and register B. At a time output from only one control circuit is activated as selected by the decoder output.

MOV A, C

This control circuit generates control signals which activate input enable signal for register A and output enable signal for register C. So the data from register C is copied into register A.

MOV B, C

This control circuit generates control signals which activate input enable signal for register B and output enable signal for register C. So the data from register C is copied into register B.

ADD

This control circuit generates control signals required for addition operation which is to be performed by the ALU i.e. $A + B \rightarrow C$.

SUB

This control circuit generates control signals required for subtraction operation which is to be performed by the ALU. i.e. $A - B \rightarrow C$.

AND

This control circuit generates control signals required for logical AND operation which is to be performed by the ALU.

OR

This control circuit generates control signals required for logical OR operation which is to be performed by the ALU.

LOAD A

This control circuit generates control signals which activate input enable signal for register A. It also indicates switch control circuit to change switch position to A.

LOAD B

This control circuit generates control signals which activate input enable signal for register B. It also indicates switch control circuit to change switch position to A.

1.1.1.8 Switch Control Circuit

Switch control circuit is responsible for switch position. It gets the input from LOAD A and LOAD B control circuits. If any of the above inputs is activated, it changes switch position to A, otherwise it holds switch at position B.

1.1.1.9 ALU

ALU takes the input from register A and register B. This input is processed according to the operation selected by the control unit. The process result is stored in the register C.

1.1.1.10 Operation

As mentioned earlier, counter starts from zero. When counter output is zero, decoder selects the first register (R_0) from the register array and the data from the selected register (R_0) is available on the common bus. Initially switch is at position B, so data from the common bus is decoded. According to data, control unit selects the operation and ALU performs the operation.

The important thing in this process is the data from selected register decides the operation. The data which decides the operation is called "Operation code" or "Opcode". Each operation has its own opcode. The Table 1.2 shows the opcodes for different operations.

For operations MOV A, C and MOV B, C the data is transferred from register C to register A and register B respectively.

In operations ADD, SUB, AND and OR data which is to be processed is taken from A and B registers. But in case of LOAD A and LOAD B operations data from common bus is directly loaded into the A and B register respectively. For this operation switch must be in position A.

| Operation | Opcode | | |
|-----------|----------------|----------------|----------------|
| | D ₂ | D ₁ | D ₀ |
| MOV A, C | 0 | 0 | 0 |
| MOV B, C | 0 | 0 | 1 |
| ADD | 0 | 1 | 0 |
| SUB | 0 | 1 | 1 |
| AND | 1 | 0 | 0 |
| OR | 1 | 0 | 1 |
| LOAD A | 1 | 1 | 0 |
| LOAD B | 1 | 1 | 1 |

Table 1.2

After each operation counter is incremented by one, so that decoder selects the next register. Again data from selected register is available on the common bus and the process is repeated. The following example will help you to understand the process sequence.

Example : Add two numbers

Steps : Load 1st number in A
Load 2nd number in B
ADD two numbers

First step is to load the number in register A. To do this, it is necessary to have opcode of LOAD A operation in the register 0 (R₀) and the number in the register 1 (R₁). Similarly in the second step it is necessary to have opcode of Load B operation in register 2 (R₂) and the number in register 3 (R₃). Finally, it is necessary to have opcode of ADD operation in register 4 (R₄).

Fig. 1.5 shows the register array contents if number 1 = 100 and number 2 = 010

Operation :

STEP 1 : Counter output (0000)

| | | | |
|-------|---|---|---|
| R_0 | 1 | 1 | 0 |
| R_1 | 1 | 0 | 0 |
| R_2 | 1 | 1 | 1 |
| R_3 | 0 | 1 | 0 |
| R_4 | 0 | 1 | 0 |
| R_5 | — | | |
| R_6 | — | | |

Initially, the counter is in the reset condition so counter output is zero (0000). The decoder A will select the register 0 (R_0), since the input for decoder is zero. Once the register 0 (R_0) is selected, data from register 0 i.e. 1 1 0 is available on the common bus. The decoder B will use this data to select the operation. As data is 1 1 0, decoder B will select LOAD A operation. LOAD A operation will enable input for register A and change the switch position from B to A. (Refer Fig. 1.6).

Fig. 1.5

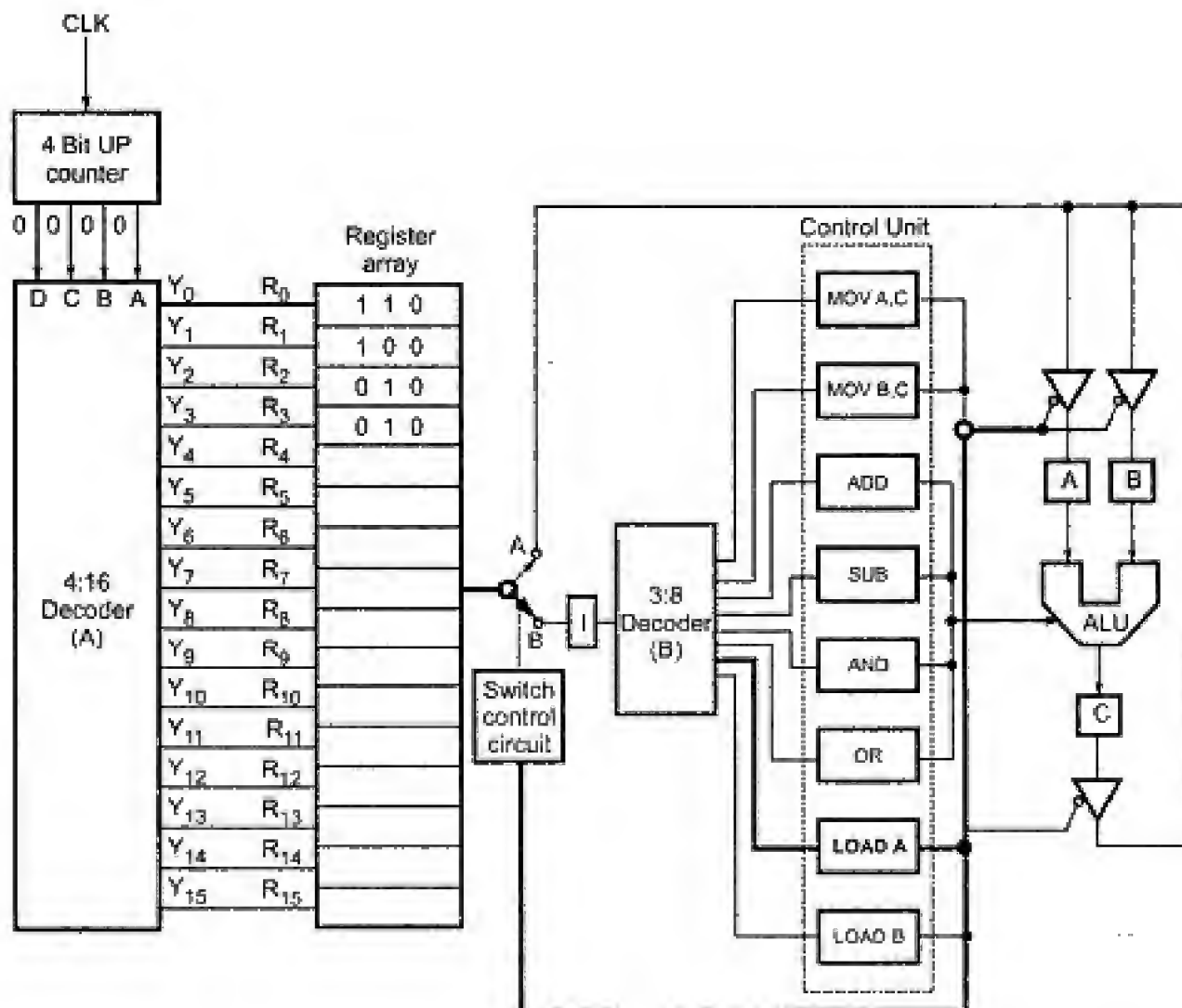
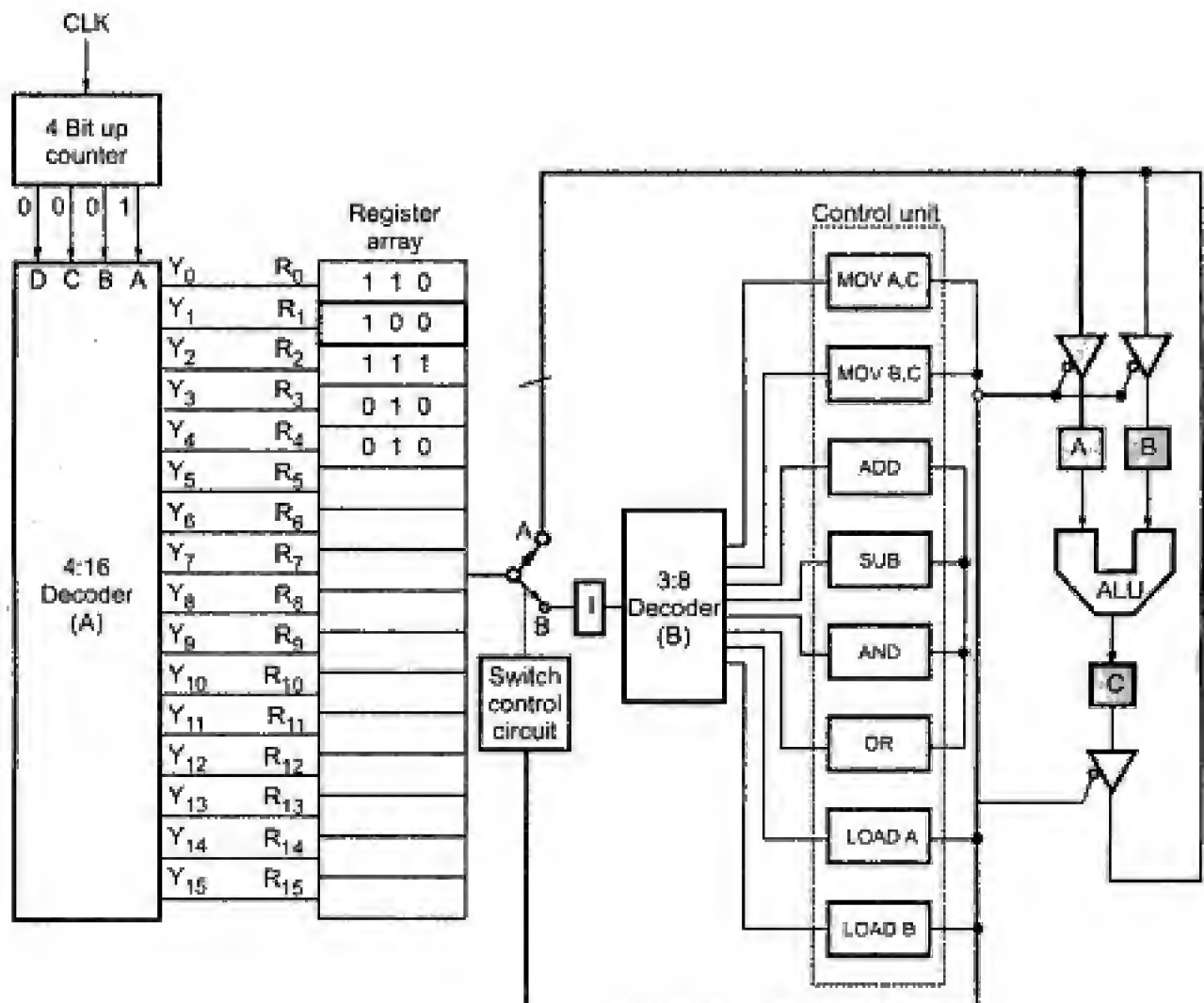


Fig. 1.6

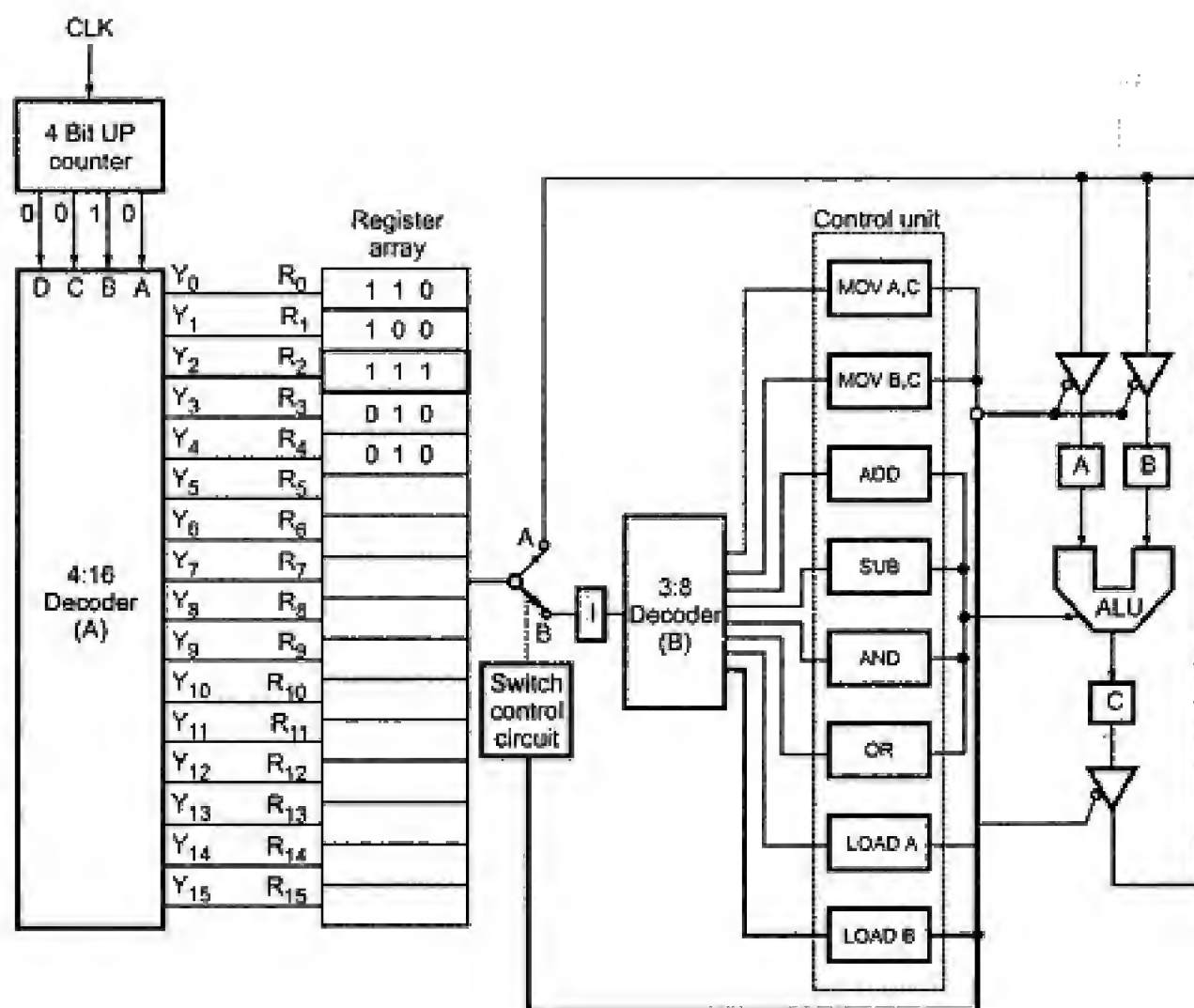
STEP 2 : Counter output (0001)

The decoder A will select the register 1 (R_1), since the input for decoder is one. Once the register 1 is selected, data from register 1 i.e. 100 is available on the common bus. This data is directly transferred to the register A because switch is positioned at A and input for register A is already enabled. After data transfer, switch position is changed from position A to position B. (Refer Fig. 1.7).

**Fig. 1.7**

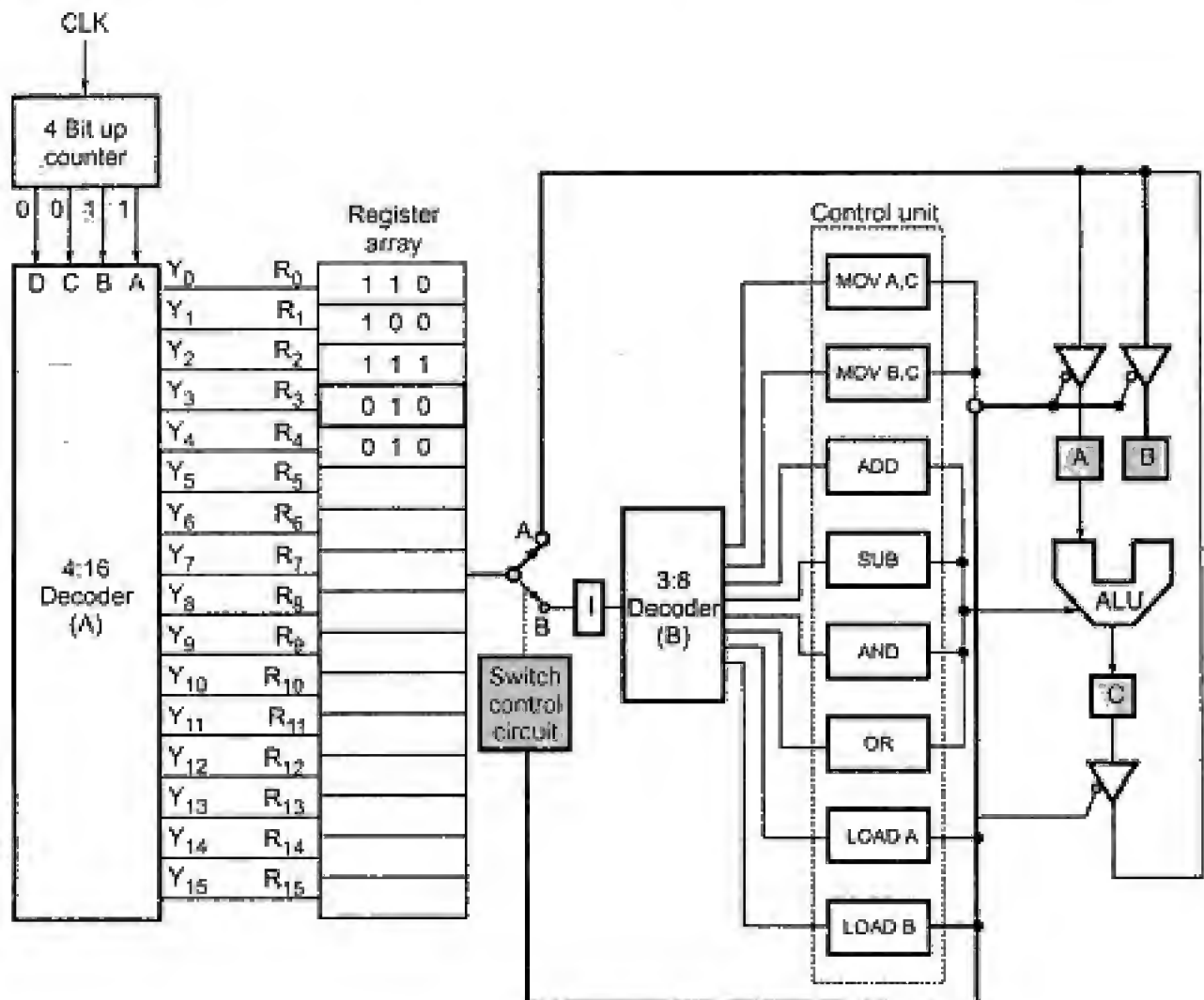
STEP 3 : Counter output (0010)

The decoder A will select the register 2 (R_2). Once the register 2 (R_2) is selected, data from register 2 i.e. 1 1 1 is available on the common bus. The decoder B will use this data to select the operation. As data is 1 1 1, decoder B will select LOAD B operation. LOAD B operation will enable input for register B and change the switch position from B to A. (Refer Fig. 1.8).

**Fig. 1.8**

STEP 4 : Counter output (0011)

The decoder A will select the register 3 (R_3). Once the register 3 is selected, data from register 3 i.e. 0 1 0 is available on the common bus. This data is directly transferred to the register B because switch is positioned at A and input for register B is already enabled. After data transfer, switch position is changed from position A to position B. (Refer Fig. 1.9)

**Fig. 1.9****STEP 5 : Counter output (0100)**

The decoder A will select the register 4 (R_4). Once the register 4 is selected, data from register 4 i.e. 0 1 0 is available on the common bus. As data is 0 1 0, decoder B will select ADD control circuit. ADD operation will generate signals to select add operation which is to be performed by ALU. Then ALU will add the contents of register A and register B and it will store the result in the register C i.e. 1 1 0 Refer Fig. 1.10.

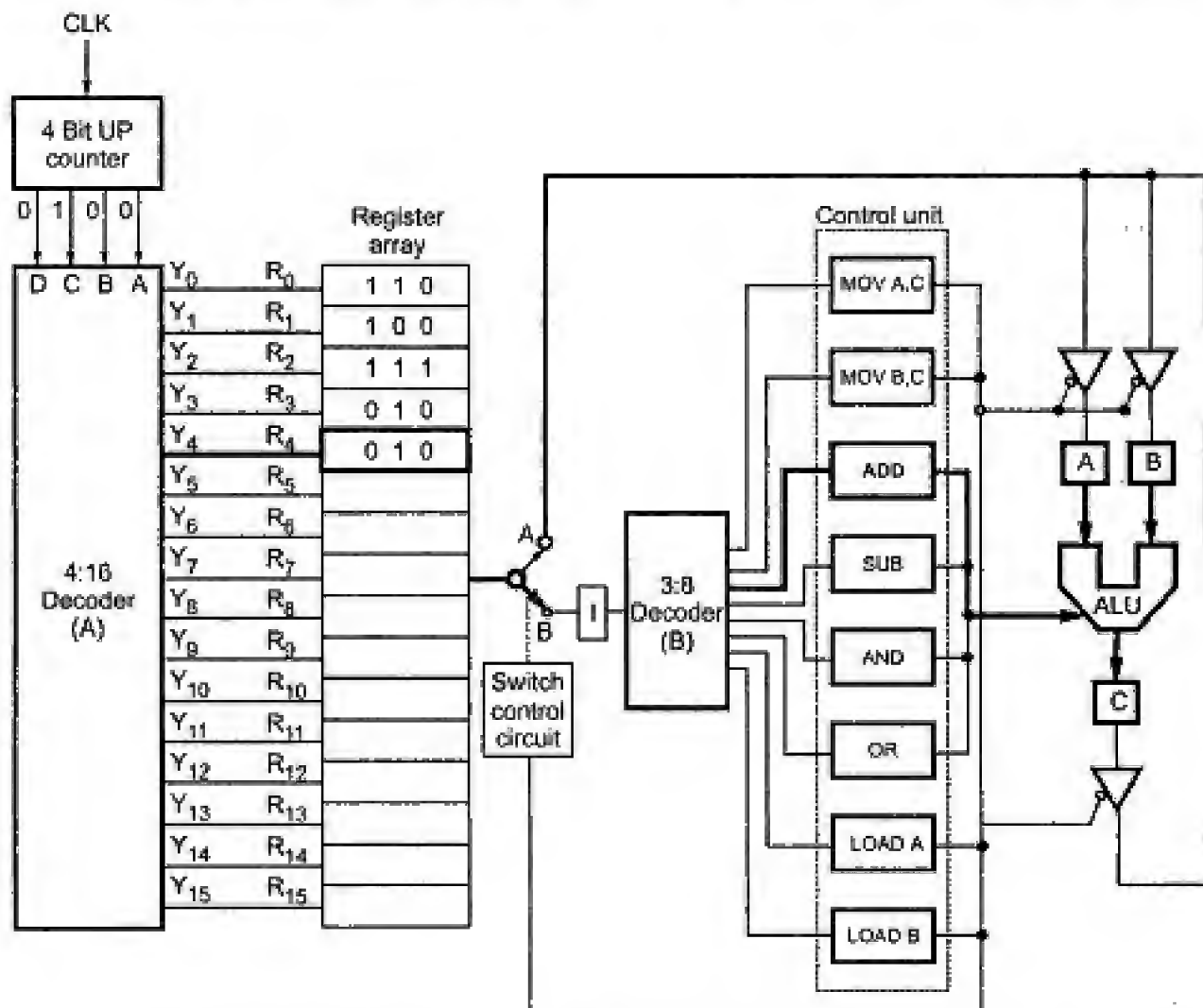


Fig. 1.10

The above example shows how two numbers can be added using the sequential circuit. Using same circuit we can subtract, multiply, divide, logically AND or logically OR the two numbers. This can be achieved by writing appropriate opcodes with necessary data in proper sequence into the register array.

Another important thing you must be noticed that LOAD A and LOAD B operations require two steps. In step one, register input is enabled and switch position is changed. In the second step actual data is moved from the next register. This data is not the operation code/opcode, but it is called **operand**. It is important to note that, to complete one operation, opcode or sometimes opcode and operand are required. This complete operation is referred as **instruction**. Thus instruction is a combination of opcode and operand.

Instruction sequences for different operations.

a) Subtract 0 1 0 (2) from 1 1 1 (7)

| Register Array | | | Operation |
|----------------|---|---|--------------------------------------|
| 1 | 1 | 0 | Load A register |
| 1 | 1 | 1 | With 7 (1 1 1) |
| 1 | 1 | 1 | Load B register |
| 0 | 1 | 0 | With 2 (0 1 0) |
| 0 | 1 | 1 | Subtract contents of B register from |
| | | | the contents of A register and store |
| | | | the result in register C |

b) Logically AND 0 1 1 (3) with 1 0 1 (5)

| Register Array | | | Operation |
|----------------|---|---|--|
| 1 | 1 | 0 | Load A register |
| 0 | 1 | 1 | With 3 (0 1 1) |
| 1 | 1 | 1 | Load B register |
| 1 | 0 | 1 | With 5 (1 0 1) |
| 1 | 0 | 0 | Logically AND the contents of Register A and register B and Store the result in register C |

c) Solve 1 0 1 (5) + 0 0 1 (1) – 0 1 1 (3)

| Register Array | | | Operation |
|----------------|---|---|--|
| 1 | 1 | 0 | Load register A |
| 1 | 0 | 1 | With 5 (1 0 1) |
| 1 | 1 | 1 | Load register B |
| 0 | 0 | 1 | With 1 (0 0 1) |
| 0 | 1 | 0 | Add contents of register A with the contents of register B and store the result in register C. |
| 0 | 0 | 0 | Copy the contents of register C to register A |
| 1 | 1 | 1 | Load register B |
| 0 | 1 | 1 | With 3 (0 1 1) |
| 0 | 1 | 1 | Subtract Contents of register B from the contents of register A and store the result in register C. |

d) Solve : 100 (4) \wedge 0 1 1 (3) \vee 0 0 1 (1)

| Register Array | | | Operation |
|----------------|---|---|---|
| 1 | 1 | 0 | Load register A |
| 1 | 0 | 0 | With 4 (100) |
| 1 | 1 | 1 | Load register B |
| 0 | 1 | 1 | With 3 (011) |
| 1 | 0 | 0 | Logically AND the contents of register A and register B and store the result in register C. |
| 0 | 0 | 0 | Copy the contents of register C to register A |
| 1 | 1 | 1 | Load register B |
| 0 | 1 | 1 | With 3 (0 1 1) |
| 1 | 0 | 1 | Logically OR the contents of register A and register B and store the result in register C. |

From the above description and examples you must have noticed that in simple model, hardware components are fixed, but by changing the sequence of opcodes and operands (instructions) it is possible to process different operations. Controlling the operation in a fixed hardware by changing sequence of instructions is the basic principle used in the microprocessor. The simple model used in this chapter is a very small version of the microprocessor. In the next section we will discuss the various terminologies used in the microprocessor.

1.1.2 Terminologies used In Microprocessor

Fig. 1.11 shows the small model with specific labels and special blocks. Block 1 represents a microprocessor whereas block 2 represents memory.

Microprocessor : It consists of counter, switch control circuit, decoder B, control unit, ALU and registers.

Memory : It is a combination of register array and the decoder circuit.

Program : The sequence of instructions written for specific operation is called program.

Program counter : The counter used in a small model is used to locate instructions in a proper sequence (program). Thus it is called a program counter.

Address and Address Bus : The output of the program counter is given to the memory. The internal decoder (decoder A) in the memory, decodes this output of the program counter to locate a specific register in the memory. This means that output of the program counter addresses a specific register in the memory. Thus the output of program counter is called address and the group of wires that carries this address is called address bus. Address bus shown in Fig. 1.11 is 4-bit wide. So it can address 16 (2^4) registers in the register array. The number of address lines decide the how many registers from the register array (memory locations) can be accessed. For example if there are 16 address lines then it is possible to access 2^{16} (65536) memory locations.

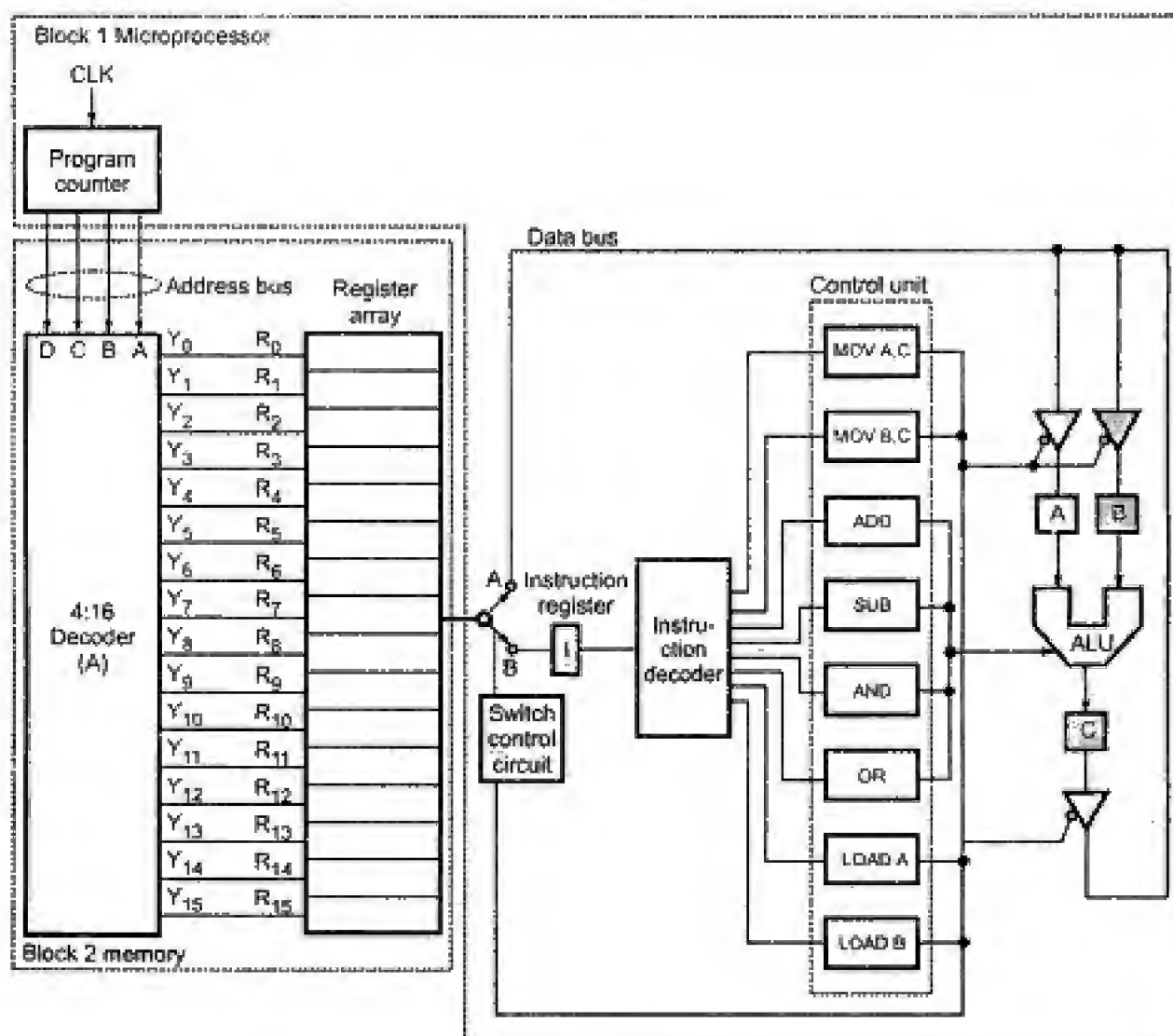


Fig. 1.11

Data and Data Bus : The data within the registers is used in different forms such as opcodes and operands. This data is moved from register to register, from memory to register through group of wires called data bus. The data bus within the microprocessor is called internal data bus whereas data bus outside the microprocessor is called external data bus. Data bus shown in the Fig. 1.11 is 3-bit wide. The size of the data bus decides

the size of operands. It also decides the maximum number of operations that microprocessor can perform. As data bus in the small model is 3-bit wide it can perform (2^3) eight different operations. The standard size of data bus is in multiple of 8 i.e. 8, 16, 24, 32 etc. The microprocessor with 8-bit data bus can execute maximum 256 (2^8) instructions.

Word Length : The group of bits that microprocessor can recognize and process is called word, and microprocessors are classified according to their word length. For example, a microprocessor with an 8-bit word is known as 8-bit microprocessor and a microprocessor with an 16-bit word is known as 16 bit microprocessor.

Instruction register : The register in which instruction opcode is stored temporarily is called instruction register.

Instruction decoder : The decoder (decoder B in small model) which takes the instruction opcode from the instruction register and decodes it to generate appropriate control signals corresponding to the instruction is called instruction decoder.

1.1.3 Different Phases in the Execution Process

The three blocks in the Fig. 1.12 shows the three different phases involved in the execution process.

These are : 1) Fetch 2) Decode 3) Execute

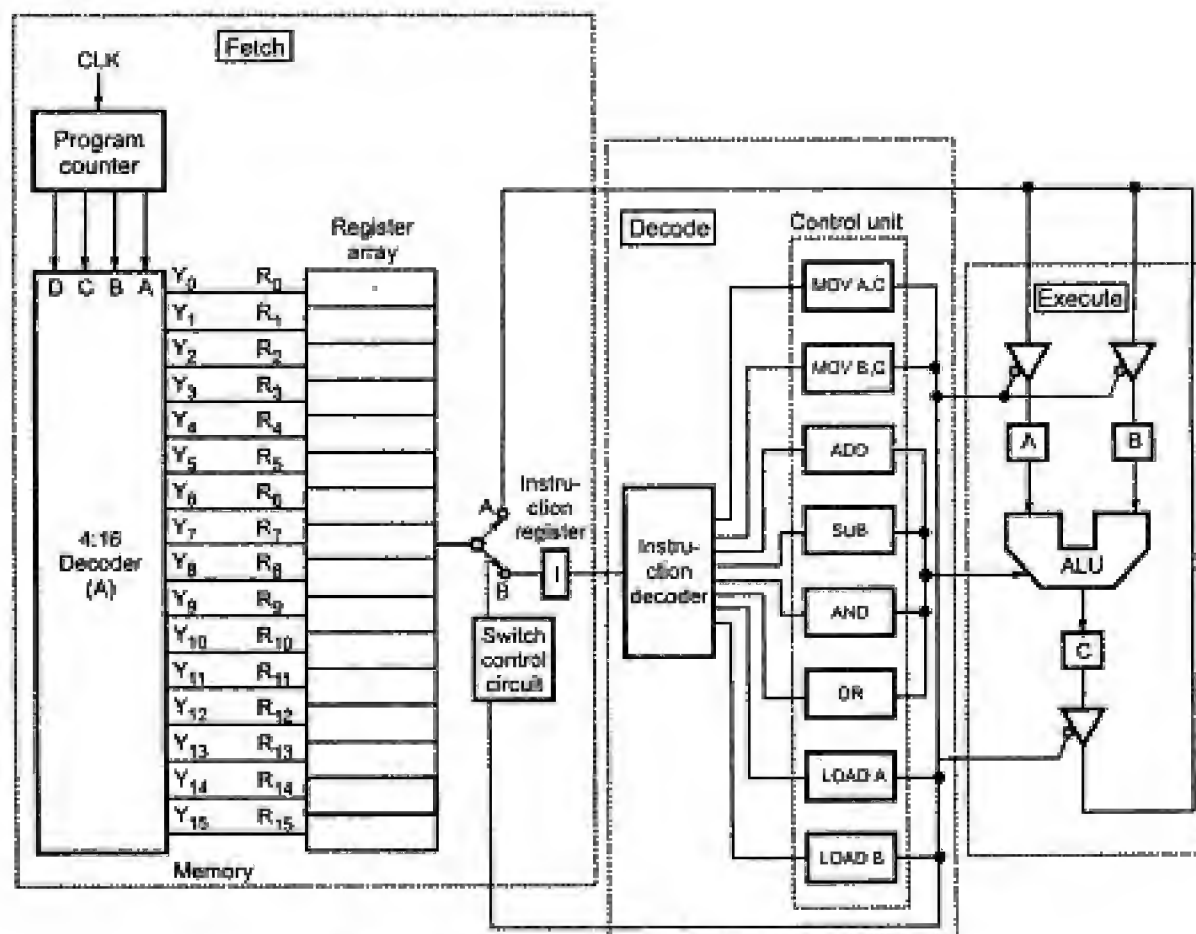


Fig. 1.12 Fetch, Decode and Execute

1.1.3.1 Fetch

In fetch phase, microprocessor places the contents of the program counter on the address bus and gets instruction code, opcode from the addressed memory location. The microprocessor then saves opcode in the instruction register.

1.1.3.2 Decode

In this phase, the opcode from the instruction register is decoded with the help of instruction decoder to generate appropriate control signals to execute the instruction.

1.1.3.3 Execute

In this phase, microprocessor generates appropriate control signals and executes the instruction.

1.1.4 Microprocessor Architecture and its Operation

In the last section we have seen simple model of microprocessor system. We have seen the important components of the microprocessor system like, microprocessor, memory unit and clock. In microprocessor block we have seen program counter, instruction decoder, instruction register, different registers, control logic and ALU. In this section, we are going to see more practical microprocessor block diagram and programmer's model. (Refer Fig. 1.13 and Fig. 1.14). The microprocessor's block diagram and the microprocessor

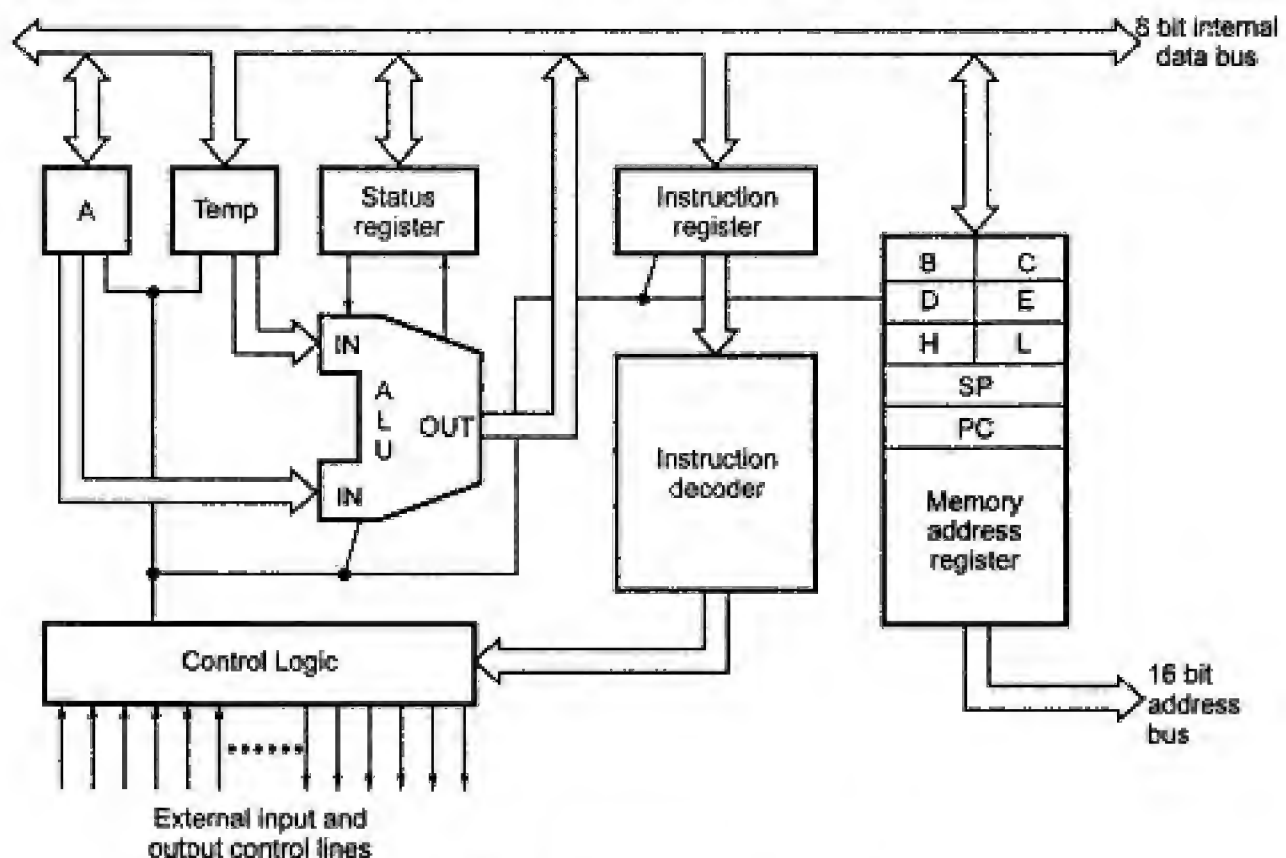


Fig. 1.13 Block diagram of microprocessor

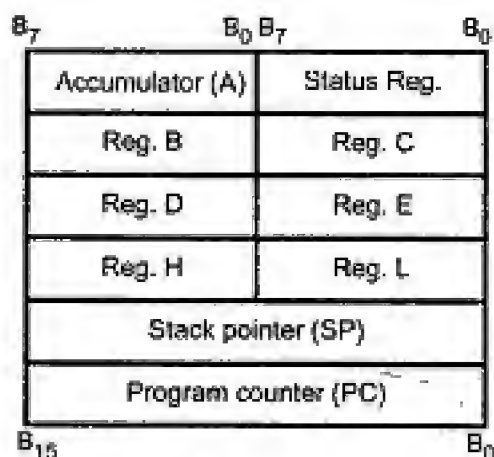


Fig. 1.14 Programmer's model

programming model show you how a specific microprocessor is constructed. The block diagram shows microprocessor's functions for data processing and data handling. It also shows how each of these logic functions are connected together. The programming model assists you in the programming process. The difference is that the programming model shows only those parts of the microprocessor which the programmer can change. So we can say that block diagram makes it easier to understand the architecture of a microprocessor and the programming model makes it easier to understand the working of microprocessor in a programming environment.

The block diagram shown in Fig. 1.13 includes three major logic devices

- ALU
- Several registers
- Control unit

The internal data bus is used to transmit data between these logic devices.

ALU :

One of the microprocessor's major logic devices is the arithmetic logic unit (ALU). It contains the microprocessor's data processing logic. It has two inputs and an output. The internal data bus of microprocessor is connected to the two inputs of ALU through the temporary register and the accumulator.

The ALU's single output is connected to the internal data bus. It allows to send the output of ALU over the bus to any device connected to the bus. In most of the microprocessors register A gives data for the ALU and after performing the operation, the resulting data word is sent to the register A and stored there. This special register, where the result is accumulated is commonly known as accumulator.

The ALU works on either one or two data words, depending on the kind of operation. The ALU uses input ports as necessary. For example, addition operation uses both ALU inputs while complementing data operation uses only one input. To complement the data word, all the bits of the word's that are logic 1 are set to logic 0, and all the bits of the word at logic 0 are set to logic 1.

The ALU of the most of the microprocessors can perform following functions.

- Add
- Subtract

- AND
- OR
- Exclusive OR
- Complement
- Shift right
- Shift left
- Increment
- Decrement

Registers :

Registers are a prominent part of the block diagram and the programming model of any microprocessor. The basic registers found in most of the microprocessors are the accumulator, the program counter, the stack pointer, the status register, the general purpose registers, the memory address register, the instruction register and the temporary data registers.

The Accumulator :

The accumulator is the major working register of microprocessor. Most of the time it is used to hold the data for manipulation. Whenever the operation processes two words, whether arithmetically or logically, the accumulator contains one of the words. The other word may be present in another register or in a memory location. Most of the times the result of an arithmetic or logical operation is placed in the accumulator. In such cases, after execution of instruction original contents of accumulator are lost because they are overwritten.

The accumulator is also used for data transfer between an I/O port and a memory location, or between one memory location and another.

The Program Counter :

The Program Counter is one of the most important registers in the microprocessor. As mentioned earlier, a program is a series of instructions stored in the memory. These instructions tell the microprocessor exactly how to solve a problem. It is important that these instructions must be executed in a proper order to get the correct result. This sequence of instruction execution is monitored by the program counter. It keeps track of which instruction is being used and what the next instruction will be.

The program counter gives the address of memory location from where the next instruction is to be fetched. Due to this the length of the program counter decides the maximum program length in bytes. For example, microprocessor that has 16 bit program counter, can address 2^{16} bytes (64 K) of memory.

Before the microprocessor can start executing a program, the program counter has to be loaded with valid memory address. This memory location must contain the opcode of first instruction in the program. In most of the microprocessors this location is fixed. For example, memory address (0000H) for 16 bit program counter. The fixed address is loaded into the program counter by resetting the microprocessor.

As said earlier, the instructions must be executed in a proper order to get the correct result. This does not mean that every instruction must follow the last instruction in the memory. But it must follow the logical sequence of the instructions. In some situations, it is better to execute part of a program that is not in sequence (don't confuse with logical sequence) with the main program. For example, there may be a part of a program that must be repeated many times during the execution of the entire program. Rather than writing repeated part of the program again and again, the programmer can write that part only once. This part is written separately. The part of the program which is written separately is called **subroutine**. The Fig. 1.15 shows how the main and subroutine programs are executed.

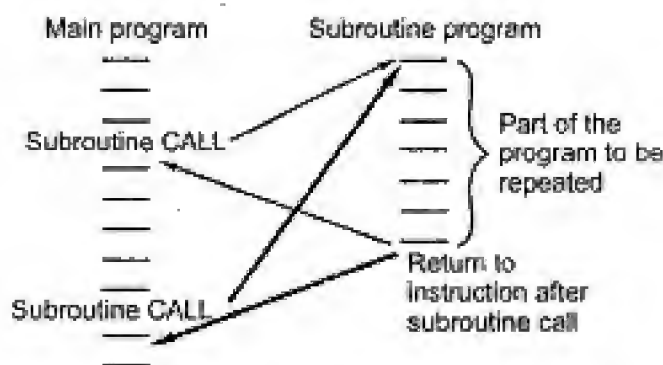


Fig. 1.15 Execution of subroutine programs

The program counter[®] does the major role in subroutine execution as it can be loaded with required memory address. With the help of instruction it is possible to load any memory address in the program counter. When subroutine is to be executed, the program counter is loaded with the memory address of the first instruction in the subroutine. After execution of the subroutine, the program counter is loaded with the memory address of the next instruction from where the program control was transferred to the subroutine program.

The Status Register : The status register is used to store the results of certain condition when certain operations are performed during execution of the program. The status register is also referred to as **flag register**. ALU operations and certain register operations may set or reset one or more bits in the status register. Status bits lead to a new set of microprocessor instructions. These instructions permit the execution of a program to change flow on the basis of the condition of bits in the status register. So the condition bits in the status register can be used to take logical decisions within the program. Some of the common status register bits are:

1) **Carry/Borrow** : The carry bit is set when the summation of two 8 bit numbers is greater than 1111 1111 (FFH). A borrow is generated when a large number is subtracted from a smaller number.

2) **Zero** : The zero bit is set when the contents of register are zero after any operation. This happens not only when you decrement the register, but also when any arithmetic or logical operation causes the contents of register to be zero.

3) **Negative or sign** : In 2's complement arithmetic, the most significant bit is a sign bit. If this bit is logic 1, the number is negative number, otherwise a positive number. The negative bit or sign bit is set when any arithmetic or logical operation gives a negative result.

4) **Auxiliary Carry** : The auxiliary carry bit of status register is set when an addition in the first 4 bits causes a carry into the fifth bit. This is often referred as half carry or intermediate carry. This is used in the BCD arithmetic.

5) **Overflow Flag** : In 2's complement arithmetic, most significant bit is used to represent sign and remaining bits are used to represent magnitude of a number (see Fig. 1.16). This flag is set if the result of a signed operation is too large to fit in the number of bits available (7-bits for 8-bit number) to represent it.

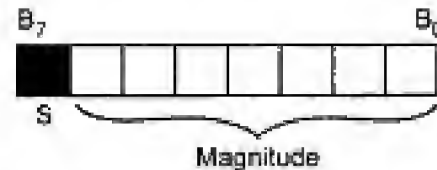


Fig. 1.16 2's Complement 8-bit number

For example, if you add the 8-bit signed number 01110110 (+118 decimal) and the 8-bit signed number 00110110 (+ 54 decimal). The result will be 10101100 (+ 172 decimal), which is the correct binary result, but in this case it is too large to fit in the 7-bits allowed for the magnitude in an 8-bit signed number. The overflow flag will be set after this operation to indicate that the result of the addition has overflowed into the sign bit.

6) **Parity** : When the result of an operation leave the indicated register with an even number of 1s, parity bit is set.

The Stack Pointer :

This is an important register which programmer uses frequently. In the earlier sections we have seen how subroutines are executed by changing the program counter contents. But one question you may have in your mind is that how the program counter is loaded with the address of the next instruction (return address) from where the program control was transferred to the subroutine. This return address is kept in a special memory area called the **stack**. Before transferring the program control to the subroutine the return address is pushed onto the stack. After the execution of subroutine the return address is popped off from the stack and loaded into the program counter.

The memory address of the stack area is given by a special register called the **stack pointer**. Like the Program Counter, the Stack Pointer automatically points to the next available location in the memory. In most microprocessors, the stack pointer decrements (points to the next lower memory address) when data is pushed on the stack. This allows the programmer to build the stack down in memory as shown in the Fig. 1.17. Usually stack operations are 2 byte operations. This means that the stack pointer decrements by two memory address locations each time when 2 byte data is pushed on the stack. When the data is popped off from the stack, the stack pointer is incremented by two memory address locations.

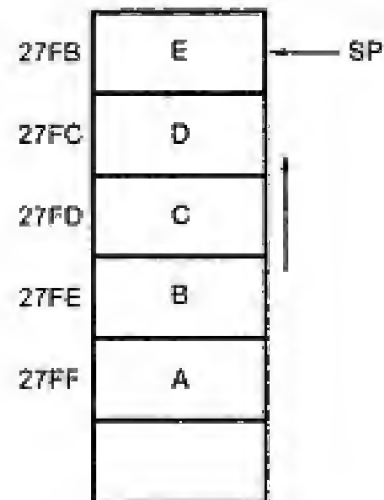


Fig. 1.17 Stack operation

It is important to note that as you go on storing (pushing) data on the stack, the stack pointer always points the last data placed on the stack and when you try to remove (pop) data you always get the last data placed on the stack. This kind of stack operation is called **LIFO (last in first out) operation**.

General Purpose Registers :

In addition to the six basic registers, most microprocessors have other registers called **general purpose registers**. The general purpose registers are used as simple storage area, mainly these are used to store intermediate results of the operation. Getting the operand from the general purpose registers is more faster than from memory so it is better to have sufficient number of general purpose register in the microprocessor. The microprocessor used in this chapter has six general purpose registers (Refer Fig. 1.13) called the B, C, D, E, H, and L registers. These registers individually can operate as 8 bit registers. Together, the BC, DE, and HL registers can operate as 16 bit register pairs.

Memory Address Register :

The memory address register gives the address of memory location that the processor wants to use. That is, memory address register holds 16-bit binary number. The output of the memory address register drives the 16-bit address bus. This output is used to select a memory location.

The Instruction Register :

The instruction register holds the operation code (opcode) of the instruction the microprocessor is currently executing. The instruction register is loaded during the opcode fetch cycle. The contents of the instruction register is used to drive the part of the control logic known as the **instruction decoder**.

Temporary Data Register :

The need for the temporary data registers arises because the ALU has no storage of its own. The ALU has two inputs. One input is supplied by accumulator and other from temporary data register. The programmer cannot access this temporary data register and, therefore it is not a part of programming model.

Control Logic :

The control logic is a important block in the microprocessor. The control logic is responsible for working of all other parts of the microprocessor together. It maintains the synchronization in operation of different parts in the microprocessor. The synchronization is achieved with the help of one of the control logic's major external inputs, microprocessor's clock. The clock is a signal which is the basis of all the timings inside the microprocessor.

Usually microprocessor's control logic is microprogrammed. This means that the architecture of the control logic itself is much like the architecture of a very special purpose microprocessor.

The control logic receives the signal from instruction decoder which decodes the instruction stored in the instruction register. The control logic then generates the control signals necessary to carry out this instruction. The control logic does a few other special functions. It looks after the microprocessor power-up sequence. It also processes interrupts. An interrupt is like a request to the microprocessor from other external devices such as the memory and I/O. The interrupt asks the microprocessor to execute a special program.

Internal Data Bus :

The internal data bus connects the different parts of microprocessor together and it enables the communication between these parts. The data transfer through this internal data bus is controlled by control logic.

Microprocessor's internal data bus usually connected to an external data bus. Due to this microprocessor can communicate with external memory or I/O devices. Usually the internal data bus is connected to the external data bus by logic called a bi-directional bus (transceiver).

1.2 Microcomputer Systems

In the last section we have seen simple model of microprocessor. But in practice, the microprocessors have more memory access capacity, more data bus width and more instructions which they can process. These factors decide which microprocessor is superior than the other. Normally, microprocessor is selected as per the requirement of the application. To have a better performance we can either go for superior microprocessor or we can use more than one microprocessor in the same system and execute the task in parallel. According to superiority and number of microprocessors used, microprocessor systems are classified as follows :

- Microcomputers
- Minicomputers
- Mainframe computers

Microcomputers : As the name implies micro-computers are smaller computers. They contain only one Central Processing Unit. One distinguishing feature of a microcomputer is that the CPU is usually a single integrated circuit called a microprocessor.

Microcomputer is the integration of microprocessor and supporting peripherals (memory and I/O devices). The word length depends on the microprocessor used and is in the range of 8 bits to 32 bits. These type of computers are used for small industrial control, process control and where storage and speed requirements are moderate.

Minicomputers : Minicomputers are the scaled up version of the microcomputers with the moderate speed and storage capacity. These are designed to process smaller data words, typically 32 bit words. This type of computers are used for scientific calculations, research, data processing application and many other.

Mainframe Computers : Mainframe computers are implemented using two or more central processing units (CPU). These are designed to work at very high speeds with large data word lengths, typically 64 bits or greater. The data storage capacity of these computers is very high. These type of computers are used for complex scientific calculations, large data processing applications, Military defense control and for complex graphics applications (e.g. : For creating walkthroughs with the help of animation softwares).

Review Questions

1. Draw the block diagram of simple microprocessor based system and explain the function of each block.
2. Draw and explain the simple model of microprocessor.
3. Define operand and instruction.
4. List the various digital components used in the microprocessor.
5. What is memory ?
6. Define program.
7. What is the function of program counter ?
8. What do you mean by address bus and data bus ?
9. What do you mean by word length ?
10. State the function of instruction register and instruction decoder.
11. List the various phases involved in the instruction execution process.
12. Explain the operation performed in the various phases of instruction execution.
13. Explain various functions of ALU.
14. What is accumulator ?

15. *What is program counter ?*
16. *What is subroutine ?*
17. *Explain the execution of subroutine program.*
18. *What is status register ? Explain its use.*
19. *List various flags in the status register.*
20. *What is stack and stack pointer ?*
21. *Explain the operation of stack.*
22. *What do you mean by general purpose registers ?*
23. *What is the function of timing and control unit in microprocessor ?*
24. *What is microcomputer ?*
25. *What is minicomputer ?*
26. *What is mainframe computer ?*



8085 Microprocessor

2.1 Introduction

In the last chapter we have seen the basic structure of microprocessor. In this chapter we will see features, pin diagram, architecture, typical configuration, and memory interfacing for 8085A microprocessor, designed by Intel.

The 8085A is an 8-bit microprocessor suitable for a wide range of applications. It is a single-chip, NMOS device implemented with approximately 6200 transistors on a 164×222 mil chip contained in a 40-pin dual-in-line package.

2.2 Features of 8085

The features of 8085 include :

1. It is an 8-bit microprocessor i.e. it can accept, process, or provide 8-bit data simultaneously.
2. It operates on a single +5 V power supply connected at V_{CC} ; power supply ground is connected to V_{SS} .
3. It operates on clock cycle with 50% duty cycle.
4. It has on chip clock generator. This internal clock generator requires tuned circuit like LC, RC or crystal. The internal clock generator divides oscillator frequency by 2 and generates clock signal, which can be used for synchronizing external devices.
5. It can operate with a 3 MHz clock frequency. The 8085A-2 version can operate at the maximum frequency of 5 MHz.
6. It has 16 address lines, hence it can access (2^{16}) 64 Kbytes of memory.
7. It provides 8 bit I/O addresses to access (2^8) 256 I/O ports.
8. In 8085, the lower 8-bit address bus ($A_0 - A_7$) and data bus ($D_0 - D_7$) are multiplexed to reduce number of external pins. But due to this, external hardware (latch) is required to separate address lines and data lines.
9. It supports 74 instructions with the following addressing modes :
 - a) Immediate
 - b) Register
 - c) Direct
 - d) Indirect
 - e) Implied

10. The Arithmetic Logic Unit (ALU) of 8085 performs :
 - a) 8 bit binary addition with or without carry
 - b) 16 bit binary addition. c) 2 digit BCD addition.
 - d) 8-bit binary subtraction with or without borrow
 - e) 8-bit logical AND, OR, EX-OR, complement (NOT), and bit shift operations.
11. It has 8-bit accumulator, flag register, instruction register, six 8-bit general purpose registers (B, C, D, E, H and L) and two 16-bit registers (SP and PC). Getting the operand from the general purpose registers is more faster than from memory. Hence skilled programmers always prefer general purpose registers to store program variables than memory.
12. It provides five hardware interrupts : TRAP, RST 7.5, RST 6.5, RST 5.5 and INTR.
13. It has serial I/O control which allows serial communication.
14. It provides control signals ($\overline{IO/\overline{M}}$, \overline{RD} , \overline{WR}) to control the bus cycles, and hence external bus controller is not required.
15. The external hardware (another microprocessor or equivalent master) can detect which machine cycle microprocessor is executing using status signals ($\overline{IO/\overline{M}}$, S_0 , S_1). This feature is very useful when more than one processors are using common system resources (memory and I/O devices).
16. It has a mechanism by which it is possible to increase its interrupt handling capacity.
17. The 8085 has an ability to share system bus with Direct Memory Access controller. This feature allows to transfer large amount of data from I/O device to memory or from memory to I/O device with high speeds.
18. It can be used to implement three chip microcomputer with supporting I/O devices like IC 8155 and IC 8355.

2.3 Architecture of 8085

Fig. 2.1 shows the architecture of 8085.

It consists of various functional blocks as listed below :

- Registers
- Arithmetic and logic unit
- Instruction decoder and machine cycle encoder
- Address buffer
- Address/Data buffer
- Incrementer/Decrementer address latch

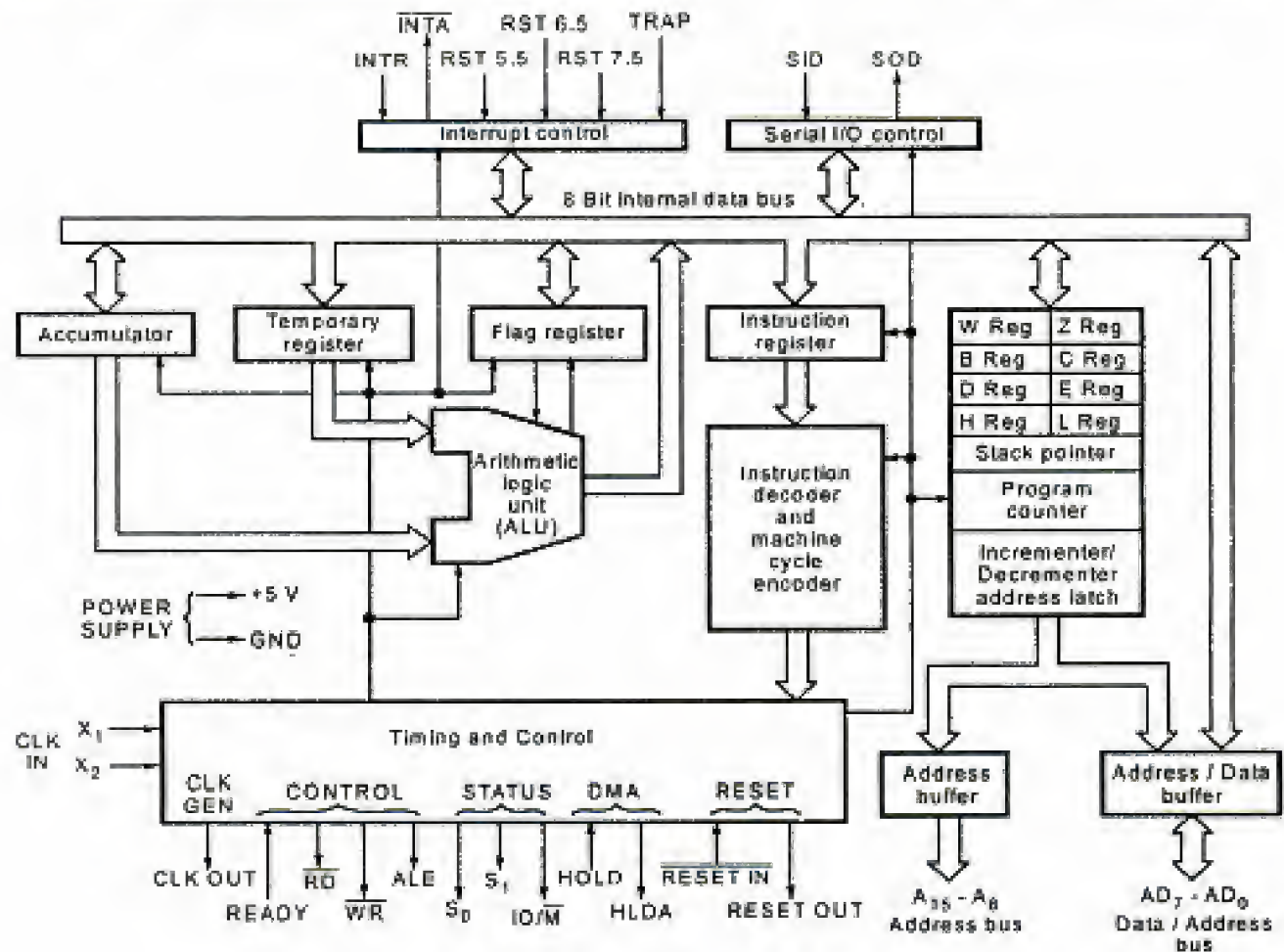


Fig. 2.1 Architecture of 8085

- Interrupt control
- Serial I/O control
- Timing and control circuitry.

2.3.1 Register Structure

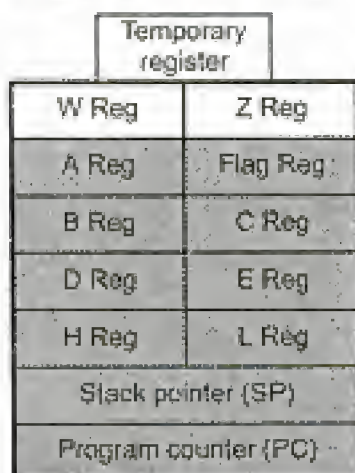


Fig. 2.2 Register structure of 8085

The Fig. 2.2 shows the register structure of 8085. The shaded portion of this register model is called **programmer's model** of 8085. It includes six 8-bit registers- (B, C, D, E, H and L) one accumulator, one flag register and two 16-bit registers (SP and PC). All these registers are accessible to programmer and hence they are included in the programmer's model. The remaining registers - temporary, W and Z are not accessible to the programmers; they are used by microprocessor for internal, intermediate operations.

The 8085 registers are classified as :

1. General Purpose Registers

2. Temporary Registers

- a) Temporary data register b) W and Z registers

3. Special Purpose Registers

- a) Accumulator b) Flag registers c) Instruction register

4. Sixteen Bit Registers

- a) Program Counter (PC) b) Stack pointer (SP)

1. General Purpose Registers :

B, C, D, E, H, and L are 8-bit general purpose registers can be used as a separate 8-bit registers or as 16-bit register pairs, BC, DE, and HL. When used in register pair mode, the high order byte resides in the first register (i.e. in B when BC is used as a register pair) and the low order byte in the second (i.e. in C when BC is used as a register pair).

HL pair also functions as a data pointer or memory pointer. These are also called **scratchpad registers**, as user can store data in them. To store and read data from these registers bus access is not required, it is an internal operation. Thus it provides an efficient way to store intermediate results and use them when required. The efficient programmer prefers to use these registers to store intermediate results than the memory locations which require bus access and hence more time to perform the operation.

2. Temporary Registers

a) **Temporary data register** : The ALU has two inputs. One input is supplied by the accumulator and other from temporary data register. The programmer can not access this temporary data register. However, it is internally used for execution of most of the arithmetic and logical instructions.

For example : ADD B is the instruction in the arithmetic group of instructions which adds the contents of register A and register B and stores result in register A. The addition operation is performed by ALU. The ALU takes inputs from register A and temporary data register. The contents of register B are transferred to temporary data register for applying second input to the ALU.

b) **W and Z registers** : W and Z registers are temporary registers. These registers are used to hold 8-bit data during execution of some instructions. These registers are not available for programmer, since 8085 uses them internally.

Use of W and Z registers :

The CALL instruction is used to transfer program control to a subprogram or subroutine. This instruction pushes the current PC contents onto the stack and loads the

given address into the PC. The given address is temporarily stored in the W and Z registers and placed on the bus for the fetch cycle. Thus the program control is transferred to the address given in the instruction. XCHG instruction exchanges the contents of H with D and L with E. At the time of exchange W and Z registers are used for temporary storage of data.

3. Special Purpose registers :

a) **Register A (accumulator)** : It is a tri-state eight bit register. It is extensively used in arithmetic, logic, load, and store operations, as well as in input/output (I/O) operations. Most of the times the result of arithmetic and logical operations is stored in the register A. Hence it is also identified as accumulator.

b) **Flag register** : It is an 8-bit register, in which five of the bits carry significant information in the form of flags : S (Sign flag), Z (Zero flag), AC (Auxiliary carry flag), P (Parity flag), and CY (carry flag), as shown in Fig. 2.3.

| D ₇ | D ₆ | D ₅ | D ₄ | D ₃ | D ₂ | D ₁ | D ₀ |
|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
| S | Z | X | AC | X | P | X | CY |

Fig. 2.3 Flag register

S-Sign flag : After the execution of arithmetic or logical operations, if bit D₇ of the result is 1, the sign flag is set. In a given byte if D₇ is 1, the number will be viewed as negative number. If D₇ is 0, the number will be considered as positive number.

Z-Zero flag : The zero flag sets if the result of operation in ALU is zero and flag resets if result is non zero. The zero flag is also set if a certain register content becomes zero following an increment or decrement operation of that register.

AC-Auxiliary carry flag : This flag is set if there is an overflow out of bit 3 i.e., carry from lower nibble to higher nibble (D₃ bit to D₄ bit). This flag is used for BCD operations and it is not available for the programmer.

P-Parity flag : Parity is defined by the number of ones present in the accumulator. After an arithmetic or logical operation if the result has an even number of ones, i.e. even parity, the flag is set. If the parity is odd, flag is reset.

CY-Carry flag : This flag is set if there is an overflow out of bit 7. The carry flag also serves as a borrow flag for subtraction. In both the examples shown below, the carry flag is set.

| ADDITION | | SUBTRACTION | |
|---------------------|--------------------|----------------------|--------------------|
| 9B H | 1001 1011 | 89 H | 1000 1001 |
| + 75 H | + 0111 0101 | - AB H | - 1010 1011 |
| Carry <u>1</u> 10 H | <u>1</u> 0001 0000 | Borrow <u>1</u> DE H | <u>1</u> 1101 1110 |

c) **Instruction register** : In a typical processor operation, the processor first fetches the opcode of instruction from memory (i.e. it places an address on the address bus and memory responds by placing the data stored at the specified address on the data bus). The CPU stores this opcode in a register called the instruction register. This opcode is further sent to the instruction decoder to select one of the 256 alternatives.

4. Sixteen Bit Registers

a) **Program counter (PC)** : Program is a sequence of instructions. As mentioned earlier, microprocessor fetches these instructions from the memory and executes them sequentially. The program counter is a special purpose register which, at a given time, stores the address of the next instruction to be fetched. Program counter acts as a pointer to the next instruction. How processor increments program counter depends on the nature of the instruction; for one byte instruction it increments program counter by one, for two byte instruction it increments program counter by two and for three byte instruction it increments program counter by three such that program counter always points to the address of the next instruction.

In case of JUMP and CALL instructions, address followed by JUMP and CALL instructions is placed in the program counter. The processor then fetches the next instruction from the new address specified by JUMP or CALL instruction. In conditional JUMP and conditional CALL instructions, if the condition is not satisfied, the processor increments program counter by three so that it points the instruction followed by conditional JUMP or CALL instruction; otherwise processor fetches the next instruction from the new address specified by JUMP or CALL instruction.

b) **Stack pointer (SP)** : The stack is a reserved area of the memory in the RAM where temporary information may be stored. A 16-bit stack pointer is used to hold the address of the most recent stack entry.

2.3.2 Arithmetic Logic Unit (ALU)

The 8085's ALU performs arithmetic and logical functions on eight bit variables. The arithmetic unit performs bitwise fundamental arithmetic operations such as addition and subtraction. The logic unit performs logical operations such as complement, AND, OR and EX-OR, as well as rotate and clear. The ALU also looks after the branching decisions.

2.3.3 Instruction Decoder

As mentioned earlier, the processor first fetches the opcode of instruction from memory and stores this opcode in the instruction register. It is then sent to the instruction decoder. The instruction decoder decodes it and accordingly gives the timing and control signals which control the register, the data buffers, ALU and external peripheral signals (explained in later sections) depending on the nature of the instruction.

The 8085 executes seven different types of machine cycles. It gives the information about which machine cycle is currently executing in the encoded form on the S_0 , S_1 and IO/\overline{M} lines. This task is done by machine cycle encoder.

2.3.4 Address Buffer

This is an 8-bit unidirectional buffer. It is used to drive external high order address bus ($A_{15} - A_8$). It is also used to tri-state the high order address bus under certain conditions such as reset, hold, halt, and when address lines are not in use.

2.3.5 Address/Data Buffer

This is an 8-bit bi-directional buffer. It is used to drive multiplexed address/data bus, i.e. low order address bus ($A_7 - A_0$) and data bus ($D_7 - D_0$). It is also used to tri-state the multiplexed address/data bus under certain conditions such as reset, hold, halt and when the bus is not in use.

The address and data buffers are used to drive external address and data buses respectively. Due to these buffers the address and data buses can be tri-stated when they are not in use.

2.3.6 Incrementer/Decrementer Address Latch

This 16-bit register is used to increment or decrement the contents of program counter or stack pointer as a part of execution of instructions related to them.

2.3.7 Interrupt Control

The processor fetches, decodes and executes instructions in a sequence. Sometimes it is necessary to have processor the automatically execute one of a collection of special routines whenever special condition exists within a program or the microcomputer system. The most important thing is that, after execution of the special routine, the program control must be transferred to the program which processor was executing before the occurrence of the special condition. The occurrence of this special condition is referred as interrupt. The interrupt control block has five interrupt inputs $RST\ 5.5$, $RST\ 6.5$, $RST\ 7.5$, $TRAP$ and $INTR$ and one acknowledge signal \overline{INTA} .

2.3.8 Serial I/O Control

In situations like, data transmission over long distance and communication with cassette tapes or a CRT terminal, it is necessary to transmit data bit by bit to reduce the cost of cabling. In serial communication one bit is transferred at a time over a single line. The 8085's serial I/O control provides two lines, SOD and SID for serial communication. The serial output data (SOD) line is used to send data serially and serial input data (SID) line is used to receive data serially.

2.3.9 Timing and Control Circuitry

The control circuitry in the processor 8085 is responsible for all the operations. The control circuitry and hence the operations in 8085 are synchronized with the help of clock signal. Along with the control of fetching and decoding operations and generating appropriate signals for instruction execution, control circuitry also generates signals required to interface external devices to the processor, 8085.

2.4 Pin Definitions of 8085

Fig. 2.4 (a) and (b) show 8085 pin configuration and functional pin diagram of 8085 respectively. The signals of 8085 can be classified into seven groups according to their functions.

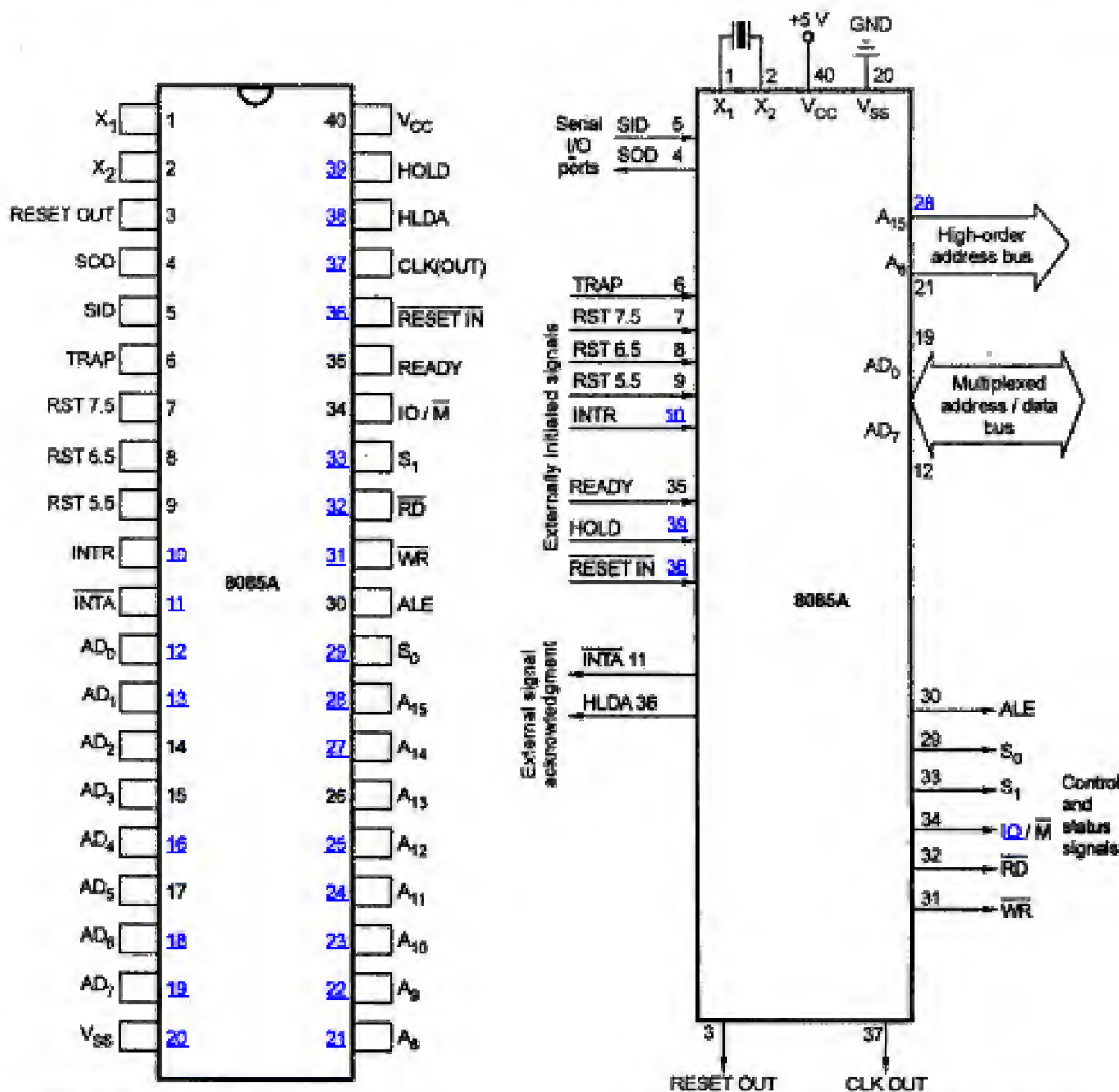


Fig. 2.4 (a) Pin configuration

Fig. 2.4 (b) Functional pin diagram

- a) Power supply and frequency signals.
- b) Data bus and address bus
- c) Control bus
- d) Interrupt signals
- e) Serial I/O signals
- f) DMA signals
- g) Reset signals

2.4.1 Power Supply and Frequency Signals

- i) V_{CC} : It requires a single +5 V power supply.
- ii) V_{SS} : Ground reference.
- iii) X_1 and X_2 : A tuned circuit like LC, RC or crystal is connected at these two pins. The internal clock generator divides oscillator frequency by 2, therefore, to operate a system at 3 MHz, the crystal of tuned circuit must have a frequency of 6 MHz.
- iv) CLK OUT : This signal is used as a system clock for other devices. Its frequency is half the oscillator frequency.

2.4.2 Data Bus and Address Bus

A) AD_0 to AD_7 : The 8 bit data bus ($D_0 - D_7$) is multiplexed with the lower half ($A_0 - A_7$) of the 16 bit address bus. During first part of the machine cycle (T_1), lower 8 bits of memory address or I/O address appear on the bus. During remaining part of the machine cycle (T_2 and T_3) these lines are used as a bi-directional data bus.

B) A_8 to A_{15} : The upper half of the 16 bit address appears on the address lines A_8 to A_{15} . These lines are exclusively used for the most significant 8 bits of the 16 bit address lines.

2.4.3 Control and Status Signals

A) ALE (Address Latch Enable) : We know that AD_0 to AD_7 lines are multiplexed and the lower half of address ($A_0 - A_7$) is available only during T_1 of the machine cycle. This lower half of address is also necessary during T_2 and T_3 of machine cycle to access specific location in memory or I/O port. This means that the lower half of an address must be latched in T_1 of the machine cycle, so that it is available throughout the machine cycle. The latching of lower half of an address bus is done by using external latch and ALE signal from 8085.

B) \overline{RD} and \overline{WR} : These signals are basically used to control the direction of the data flow between processor and memory or I/O device/port. A low on \overline{RD} indicates that the data must be read from the selected memory location or I/O port via data bus. A low on \overline{WR} indicates that the data must be written into the selected memory location or I/O port via data bus.

C) $\overline{IO/\overline{M}}$, S_0 and S_1 : $\overline{IO/\overline{M}}$ indicates whether I/O operation or memory operation is being carried out. S_1 and S_0 indicate the type of machine cycle in progress.

D) **READY** : It is used by the microprocessor to sense whether a peripheral is ready or not for data transfer. If not, the processor waits. It is thus used to synchronize slower peripherals to the microprocessor.

2.4.4 Interrupt Signals

The 8085 has five hardware interrupt signals : RST 5.5, RST 6.5, RST 7.5, TRAP and INTR. The microprocessor recognizes interrupt requests on these lines at the end of the current instruction execution.

The \overline{INTA} (Interrupt Acknowledge) signal is used to indicate that the processor has acknowledged an INTR interrupt.

2.4.5 Serial I/O Signals

A) **SID (Serial I/P Data)** : This input signal is used to accept serial data bit by bit from the external device.

B) **SOD (Serial O/P Data)** : This is an output signal which enables the transmission of serial data bit by bit to the external device.

2.4.6 DMA Signal

A) **HOLD** : This signal indicates that another master is requesting for the use of address bus, data bus and control bus.

B) **HLDA** : This active high signal is used to acknowledge HOLD request.

2.4.7 Reset Signals

A) **$\overline{RESET\ IN}$** : A low on this pin

- 1) Sets the program counter to zero (0000H).
- 2) Resets the interrupt enable and HLDA flip-flops.
- 3) Tri-states the data bus, address bus and control bus. (Note : Only during RESET is active).
- 4) Affects the contents of processor's internal registers randomly.

On reset, the PC sets to 0000H which causes the 8085 to execute the first instruction from address 0000H. For proper reset operation reset signal must be held low for at least 3 clock cycles. The power-on reset circuit can be used to ensure execution of first instruction from address 0000H.

B) **$\overline{RESET\ OUT}$** : This active high signal indicates that processor is being reset. This signal is synchronized to the processor clock and it can be used to reset other devices connected in the system.

2.5 Clock Circuits

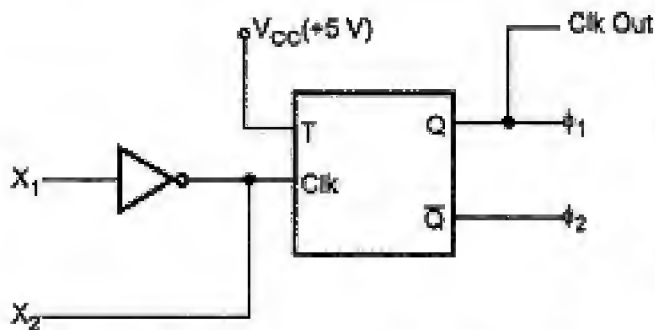


Fig. 2.5 Block diagram of built-in clock generator

The 8085 has on chip clock generator. Fig. 2.5 shows the internal block diagram of the on chip clock generator. The internal clock generator requires tuned circuit like LC, RC or crystal, or external clock source as an input to generate the clock. The internal T-flip flop divides the frequency by 2. Hence the operating frequency of the 8085 is always half of the oscillator frequency.

LC Tuned Circuit :

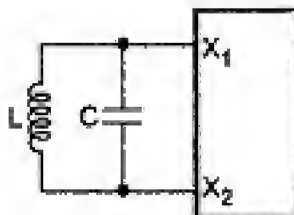


Fig. 2.6 LC circuit

It is a LC resonant tank circuit. The resonant frequency for this circuit is given by

$$f_r = \frac{1}{2\pi\sqrt{L(C_{ext} + C_{int})}}$$

Where C_{int} is the internal capacitance and it is normally 15 pF. The output frequency of this circuit has 10% variations. To minimize the variations in the output frequency, it is

recommended to have C_{ext} at least twice that of C_{int} i.e. 30 pF.

RC Tuned Circuit : Fig. 2.7 shows the RC tuned circuit. The output frequency of this circuit is also not exactly stable. But this circuit has an advantage that its component cost is less.

Crystal Oscillator Circuit :

Fig. 2.8 shows the crystal oscillator circuit. It is the most stable circuit. The 20 pF capacitor in the circuit is connected to assure oscillator start-up at the correct frequency.

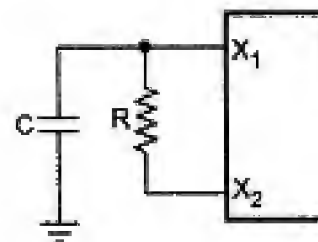


Fig. 2.7 RC circuit

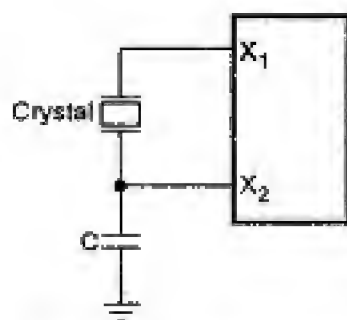


Fig. 2.8 Crystal clock circuit

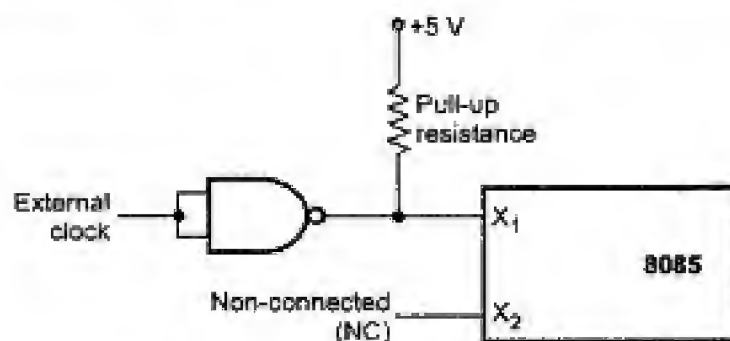


Fig. 2.9 External frequency source

External Clock :

Fig. 2.9 shows how to drive clock input of 8085 with external frequency source. Here external clock is applied at X₁ input and X₂ input is kept open.

2.6 Reset Circuit

On reset, the PC sets to 0000H which causes the 8085 to execute the first instruction from address 0000H. For proper reset operation reset signal must be held low for at least 3 clock cycles. The power-on reset circuit can be used to ensure execution of first instruction from address 0000H. Fig. 2.10 shows the power-on reset circuit with typical R, C values. (Note: R, C values may vary due to power supply ramp up time).

Upon power-up, $\overline{\text{RESET IN}}$ must remain low for at least 10ms after minimum V_{cc} has been reached, in the circuit shown in Fig. 2.10. Upon power up or key press, the $\overline{\text{RESET IN}}$ goes low and slowly rises to +5 V, providing sufficient time for the processor to reset the system. The diode is connected to discharge the capacitor immediately when power supply is switched OFF.

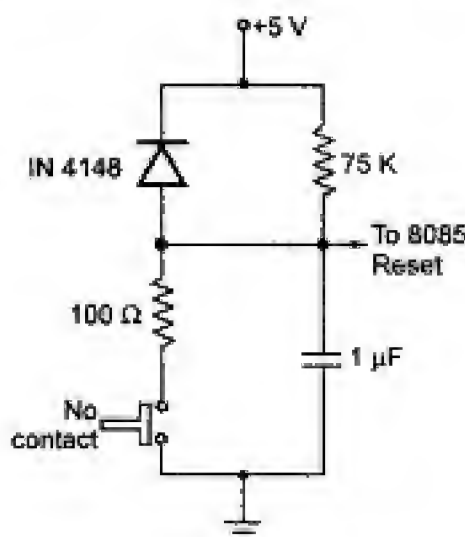


Fig. 2.10 Power on reset

After RESET, 8085 loads 0000H in PC register and clears the INTE flag. Before going to execute interrupt service routine, it is necessary to setup certain parameters, required to execute interrupt service routine. To avoid interrupt to occur before completion of these initial requirements, after power on or reset, INTE flip-flop is cleared to disable interrupts. It can be enabled by EI instruction after initial settings.

As we know that, after power up or reset 8085 fetches its first instruction from 0000H address, and it has to be the first instruction from monitor program. Therefore EPROM consisting of monitor program must be located from address 0000H in any 8085 microprocessor system.

2.7 8085 Interrupt Structure and Operation

2.7.1 Types of Interrupts

The 8085 has multilevel interrupt system. It supports two types of interrupts:

- a. Hardware b. Software

Hardware : Some pins on the 8085 allow peripheral device to interrupt the main program for I/O operations. When an interrupt occurs, the 8085 completes the instruction it is currently executing and transfers the program control to a subroutine that services the peripheral device. Upon completion of the service routine, the MPU returns to the main program. These types of interrupts, where MPU pins are used to receive interrupt requests, are called **hardware interrupts**.

Software : In software interrupts, the cause of the interrupt is an execution of the instruction. These are special instructions supported by the microprocessor. After execution of these instructions microprocessor completes the execution of the instruction it is currently executing and transfers the program control to the subroutine program. Upon completion of the execution of the subroutine program, program control returns to the main program.

2.7.2 Overall Interrupt Structure

2.7.2.1 Hardware Interrupts in 8085

The 8085 has five hardware interrupts :

- 1. TRAP 2. RST 7.5 3. RST 6.5 4. RST 5.5 5. INTR

When any of these pins, except INTR, is active, the internal control circuit of the 8085 produces a CALL to a predetermined memory location. This memory location, where the subroutine starts is referred to as **vector location** and such interrupts are called **vectored interrupts**. The INTR is not a vectored interrupt. It receives the address of the subroutine from the external device.

In 8085, all interrupts except TRAP are maskable. When logic signal is applied to a maskable interrupt input, the 8085 is interrupted only if that particular input is enabled. These interrupts can be enabled or disabled under program control. If disabled, 8085 disables an interrupt request. The interrupt TRAP is nonmaskable which means that it is not maskable by program control. The Fig. 2.11 shows the interrupt structure of 8085. The figure indicates that, the 8085 is designed to respond to edge triggering, level triggering or both.

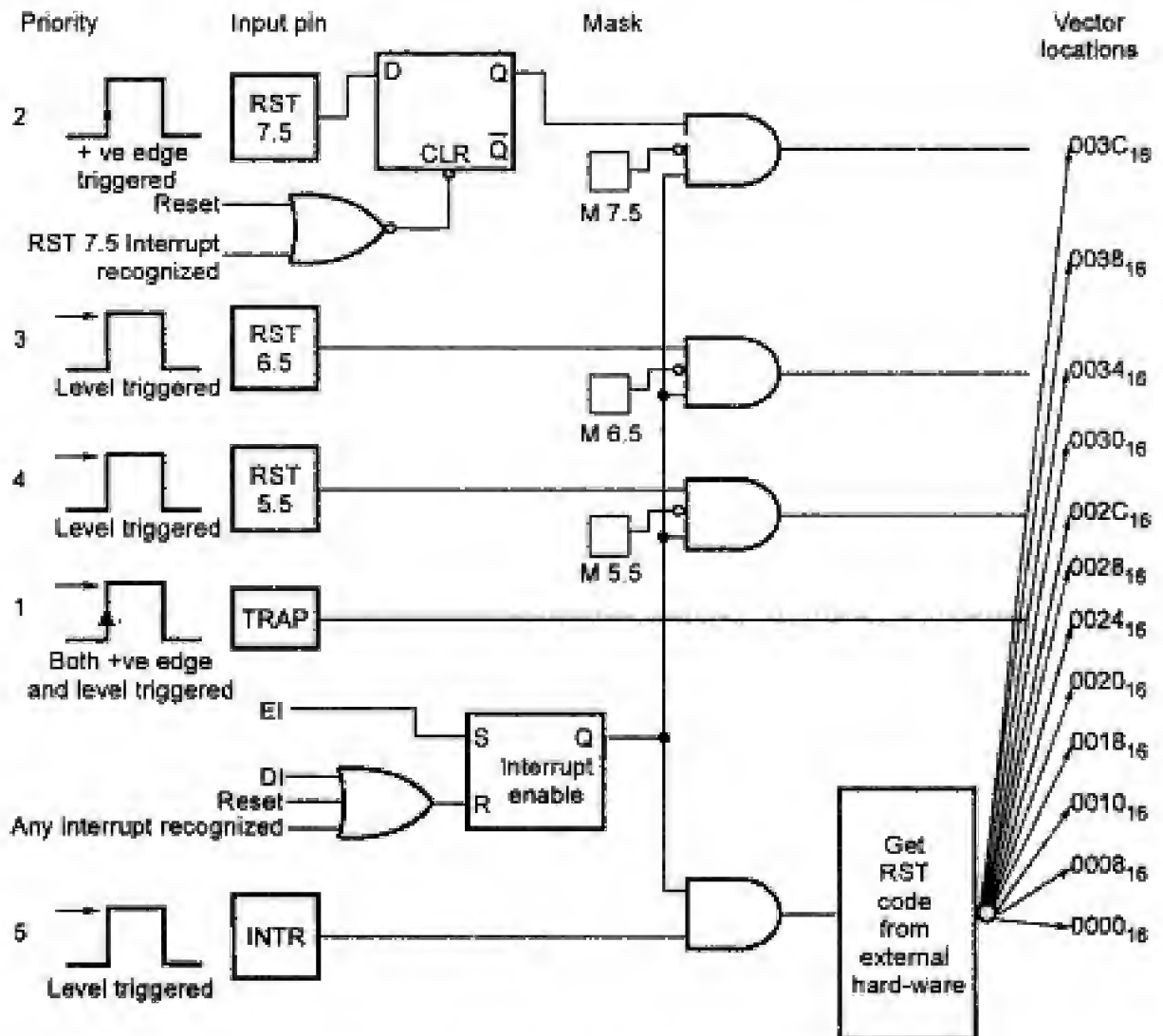


Fig. 2.11 Interrupt structure of 8085

TRAP :

This interrupt is a nonmaskable interrupt. It is unaffected by any mask or interrupt enable. TRAP has the highest priority. TRAP interrupt is edge and level triggered. This means that the TRAP must go high and remain high until it is acknowledged. This avoids false triggering caused by noise and transients.

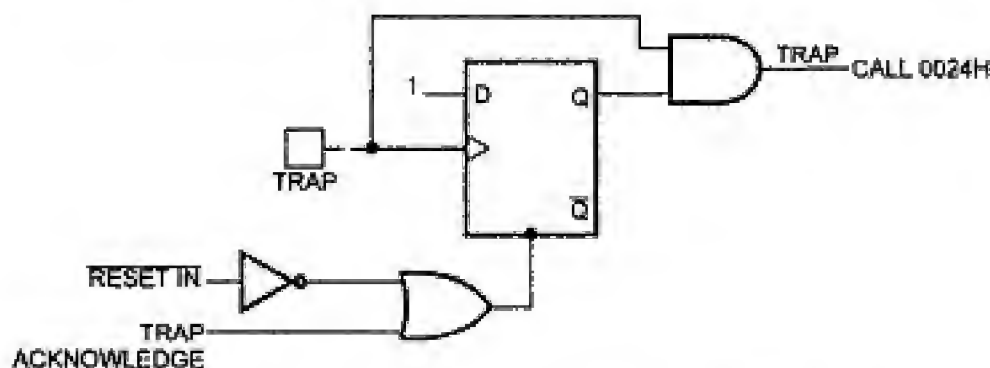


Fig. 2.12 Interrupt circuit for trap interrupt

As shown in the Fig. 2.12, the positive edge of TRAP signal sets the D flip-flop. However, due to the AND gate, it is necessary to sustain high level on the TRAP input. There are two ways to clear TRAP interrupt :

1. By resetting microprocessor i.e. giving a low signal on $\overline{\text{RESETIN}}$ pin (External signal).
2. By giving a high TRAP ACKNOWLEDGE (Internal signal).

After recognition of TRAP interrupt, 8085 internally generates a high TRAP ACKNOWLEDGE which clears the flip flop. Once the TRAP is acknowledged, the 8085 completes its current instruction. It then pushes the address of the next instruction i.e. return address onto the stack and loads PC with the fixed vector address 0024H. Due to this, 8085 starts execution of instructions from address 0024H which is the starting address of an interrupt service routine for TRAP.

RST 7.5 : The RST 7.5 interrupt is a maskable interrupt. It has the second highest priority. As shown in Fig. 2.11, it is positive edge triggered and the positive edge trigger is stored internally by the D-flip flop until it is cleared by software reset using SIM instruction or by internally generated ACKNOWLEDGE signal.

The positive edge signal on the RST 7.5 pin sets the D flip flop. If the mask bit M 7.5 is 0 i.e. RST 7.5 is unmasked then 8085 completes its current instruction. It then pushes the address of the next instruction onto the stack and loads PC with the fixed vector address 003CH. Due to this, 8085 starts execution of instructions from address 003CH which is the starting address of an interrupt service routine for RST 7.5.

RST 6.5 and RST 5.5 : The RST 6.5 and RST 5.5 both are level triggered. These interrupts can be masked using SIM instruction. The RST 6.5 has the third priority whereas RST 5.5 has the fourth priority. The vector addresses of RST 6.5 and RST 5.5 are 0034H and 002CH respectively. After recognition of RST 6.5 or RST 5.5 interrupt, 8085 completes its current instruction; pushes the address of next instruction onto the stack and loads PC with corresponding vector address.

INTR : INTR is a maskable interrupt, but not the vector interrupt. It has the lowest priority. The following sequence of events occur when INTR signal goes high.

1. The 8085 checks the status of INTR signal during execution of each instruction.
2. If INTR signal is high, then 8085 completes its current instruction and sends an active low interrupt acknowledge signal ($\overline{\text{INTA}}$) if the interrupt is enabled.
3. In response to the $\overline{\text{INTA}}$ signal, external logic places an instruction OPCODE on the data bus. In the case of multibyte instruction, additional interrupt acknowledge machine cycles are generated by the 8085 to transfer the additional bytes into the microprocessor.
4. On receiving the instruction, the 8085 saves the address of next instruction on stack and executes received instruction.

Note : Theoretically, the external logic can place any instruction code on the data bus in response to the $\overline{\text{INTA}}$. However, only CALL and RST codes save the contents of the PC on the stack and branch program control to the subroutine address.

Response for RST instruction : If the external device places an opcode for any one of the RST instruction (RST 0 - RST 7), then 8085 pushes the contents of PC onto the stack. It then branches the program control to the vector address of the corresponding RST instruction.

Response for CALL instruction : If the external device places an opcode for CALL instruction then 8085 generates two additional interrupt acknowledge cycles.

1. It sends an active low interrupt acknowledge signal second time.
2. In response to second $\overline{\text{INTA}}$ signal, external logic places the lower byte address for the CALL instruction.
3. After receiving lower byte address, 8085 sends the third interrupt acknowledge signal.
4. In response to third $\overline{\text{INTA}}$ signal, external logic places the higher byte address for the CALL instruction.
5. After receiving sixteen bit address for CALL, 8085 pushes the contents of the PC onto the stack and branches the program control to the subroutine whose address is received from the external logic.

Example : The Fig. 2.13 shows the diagram of external logic that gives the RST 7 instruction opcode on interrupt acknowledge.

External logic controls a tri-state buffer with the $\overline{\text{INTA}}$ signal in order to place an opcode for RST 7 instruction. The $\overline{\text{INTA}}$ signal from the microprocessor is used as an Output Enable signal for the buffer as well as reset signal for D flip flop. The request from the I/O device is routed through the D flip-flop to the INTR. The D flip flop is used to hold the INTR signal high until 8085 gives interrupt acknowledge signal. The $\overline{\text{INTA}}$ signal that is generated enables the tri-state buffer whose data inputs are hardwired to the value

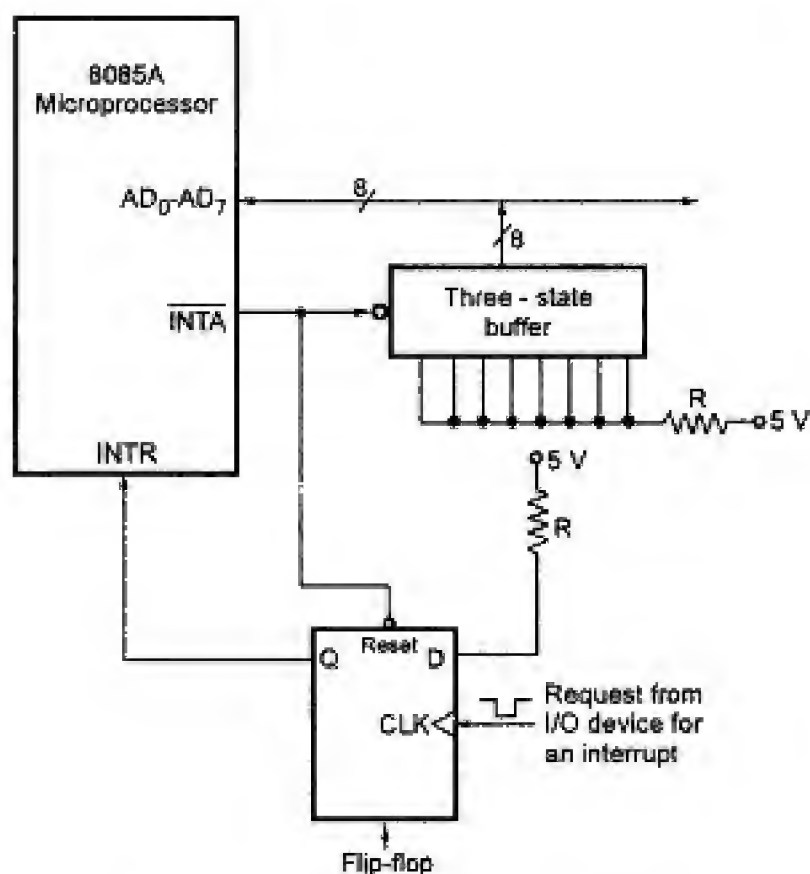


Fig. 2.13 External logic that gives the RST 7 instruction opcode

equal to the opcode for RST 7 (FFH) instruction. The 8085 receives this opcode during interrupt acknowledge cycle. After receiving the opcode 8085 pushes the contents of program counter onto the stack, thus saving the return address. It then branches the program control to the address 0038H (Vector address of RST 7). Table 2.1 shows the summary of hardware interrupts in 8085.

| Interrupt type | Trigger | Priority | Maskable | Vector address |
|----------------|----------------|---------------------------|----------|----------------|
| TRAP | Edge and Level | 1 st (Highest) | No | 0024 H |
| RST 7.5 | Edge | 2 nd | Yes | 003CH |
| RST 6.5 | Level | 3 rd | Yes | 0034H |
| RST 5.5 | Level | 4 th | Yes | 002CH |
| INTR | Level | 5 th (Lowest) | Yes | - |

Table 2.1

2.7.2.2 Software Interrupts in 8085

The 8085 has eight software interrupts from RST 0 to RST 7. The vector address for these interrupts can be calculated as follows.

$$\text{Interrupt number} \times 8 = \text{vector address}$$

For example :

$$5 \times 8 = 40 = 28\text{H}$$

\therefore Vector address for interrupt RST 5 is 0028H.

The Table 2.2 shows the vector addresses of all interrupts.

| Instruction | HEX code | Vector address |
|-------------|----------|----------------|
| RST 0 | C7 | 0000H |
| RST 1 | CF | 0008H |
| RST 2 | D7 | 0010H |
| RST 3 | DF | 0018H |
| RST 4 | E7 | 0020H |
| RST 5 | EF | 0028H |
| RST 6 | F7 | 0030H |
| RST 7 | FF | 0038H |

Table 2.2 Vector addresses for software interrupts

2.7.3 Masking / Unmasking of Interrupts

As mentioned earlier, maskable interrupts are enabled and disabled under program control. In this section we will see how interrupts can be masked or unmasked using program control. There are three instructions used for masking / unmasking of interrupts :

1. EI
2. DI
3. SIM

EI : Enable Interrupt

The EI instruction sets the interrupt enable flip-flop, as shown in Fig. 2.11. Thus RST 7.5, RST 6.5, RST 5.5 and INTR are enabled using EI instruction.

It is important to note that when any interrupt is acknowledged, interrupt enable flip flop resets and disables all interrupts. To enable interrupt in further process it is necessary to execute EI instruction within interrupt service routine.

DI : Disable Interrupt

The DI instruction resets the interrupt enable flip flop, as shown in Fig. 2.11. Thus it disables RST 7.5, RST 6.5, RST 5.5 and INTR interrupts.

SIM : Set Interrupt Mask

This instruction is used to set interrupt mask and to send serial output. It transfers the contents of accumulator to interrupt control logic and serial I/O port. Thus it is necessary to load appropriate contents in the accumulator before execution of SIM instruction.

2.7.4 Pending Interrupts

The Read Interrupt Mask, RIM, instruction is used to handle pending interrupts. It loads the status of the interrupt mask, the pending interrupts and the contents of the serial input data line, SID, into the accumulator. Thus, it is possible to monitor status of interrupt mask, pending interrupts and serial input. There are number of interrupts. When one interrupt is being serviced, other interrupt requests may occur. If the interrupt requests are of higher priority, 8085 branches program control to the requested interrupt service routines. But when the interrupt requests are of lower priority, 8085 stores the information about these interrupt requests. Such interrupts are called **pending interrupts**. The status of pending interrupts can be monitored using RIM instruction.

2.8 I/O, Memory and System Buses

We know that the central processing unit, memory unit and I/O unit are the hardware components/modules of the computer. They work together with communicating each other and have paths for connecting the modules together. The collection of paths connecting the various modules is called the **interconnection structure**. The design of this interconnection structure will depend on the exchanges that must be made between modules. A group of wires, called **bus** is used to provide necessary signals for communication between modules. A bus that connects major computer components/modules (CPU, memory, I/O) is called a **system bus**. The system bus is a set of conductors that connects the CPU, memory and I/O modules. Usually, the system bus is separated into three functional groups :

- Data Bus
- Address Bus
- Control Bus

The 8085 microprocessor provides $AD_0 - AD_{15}$ lines as an address/data lines, and IO/\overline{M} , \overline{RD} and \overline{WR} lines as control lines. We have to take support of external components to generate address, data and control bus for memory and I/O interfacing.

2.8.1 Generation of Address and Data Bus - Demultiplexing $AD_7 - AD_0$

We know that AD_0 to AD_7 lines are multiplexed and the lower half of address ($A_0 - A_7$) is available only during T_1 of the machine cycle. This lower half of address is also necessary during T_2 and T_3 of machine cycle to access specific location in memory or I/O port. This means that the lower half of an address bus must be latched in T_1 of the machine cycle, so that it is available throughout the machine cycle. The latching of lower

half of an address is done by using external latch and ALE signal from 8085. The Fig. 2.14 shows the hardware connection for latching the lower half of an address. The IC 74LS373 is an 8-bit latch, having 8 D flip-flops. The input is transferred to the output only when clock is high. This clock signal is driven by ALE signal from 8085. The ALE signal is activated only during T_1 , so input is transferred to the output only during T_1 i.e. address ($A_0 - A_7$) on the AD_0 to AD_7 multiplexed bus. In the remaining part of the machine cycle, ALE signal is disabled so output of the latch ($A_0 - A_7$) remains unchanged. To latch lower half of an address, in each machine cycle, the 8085 gives ALE signal high during T_1 of every machine cycle.

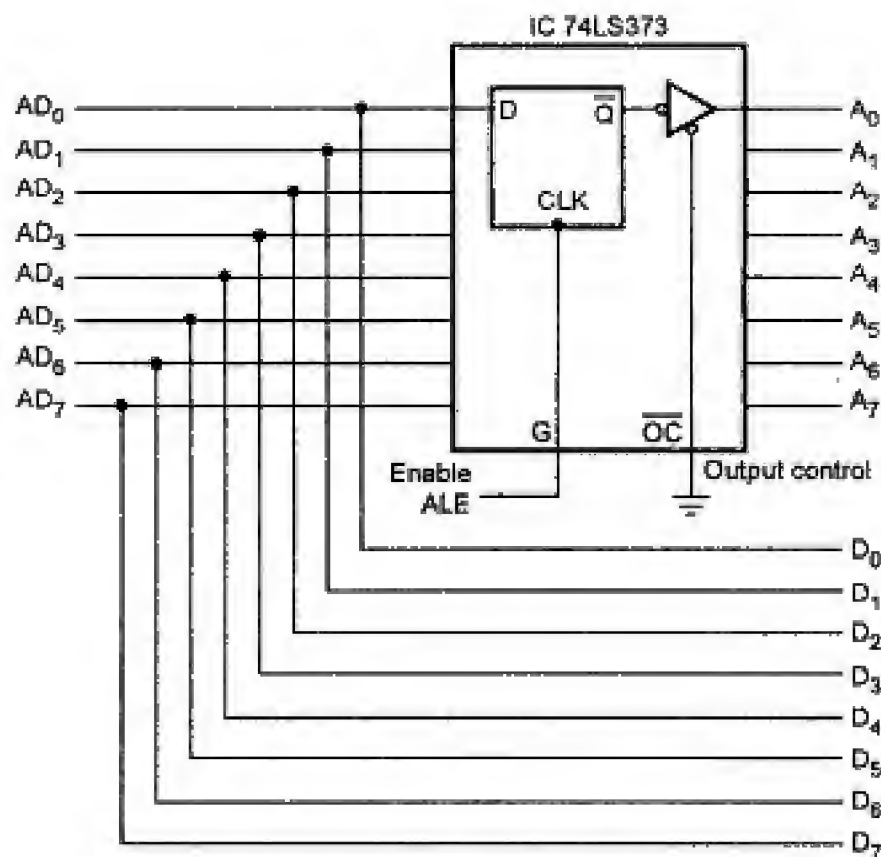


Fig. 2.14 Latching circuit

The address lines $A_8 - A_{15}$ are not multiplexed, thus the output of latch and address lines $A_8 - A_{15}$ constitutes address bus ($A_0 - A_{15}$).

2.8.2 Generation of Control Signals

The 8085 microprocessor provides \overline{RD} and \overline{WR} signals to initiate read or write cycle. Because these signals are used both for reading/writing memory and for reading/writing an input device, it is necessary to generate separate read and write signals for memory and I/O devices.

The 8085 provides $\overline{IO/\overline{M}}$ signal to indicate whether the initiated cycle is for I/O device or for memory device. Using $\overline{IO/\overline{M}}$ signal along with \overline{RD} and \overline{WR} , it is possible to generate separate four control signals :

\overline{MEMR} (Memory Read) : To read data from memory.

\overline{MEMW} (Memory Write) : To write data in memory.

\overline{IOR} (I/O Read) : To read data from I/O device.

\overline{IOW} (I/O Write) : To write data in I/O device.

Fig. 2.15 shows the circuit which generates \overline{MEMR} , \overline{MEMW} , \overline{IOR} and \overline{IOW} signals.

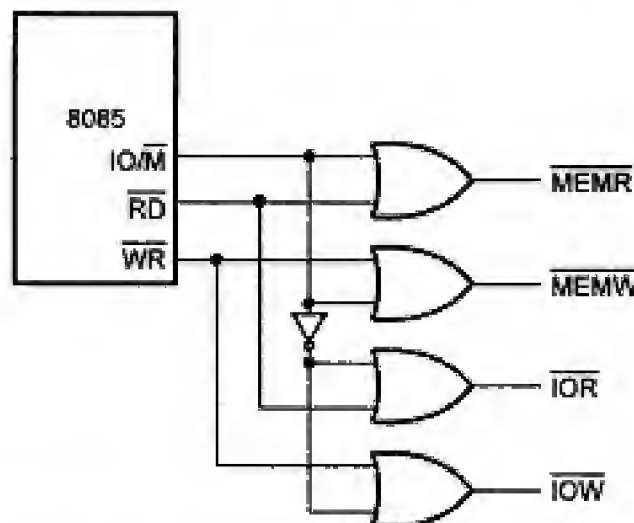


Fig. 2.15 Generation of \overline{MEMR} , \overline{MEMW} , \overline{IOR} and \overline{IOW} signals

We know that for OR gate, when both the inputs are low then only output is low. Table 2.3 shows the truth table used to generate \overline{MEMR} , \overline{MEMW} , \overline{IOR} and \overline{IOW} signals. The signal $\overline{IO/\overline{M}}$ goes low for memory operation. This signal is logically ORed with \overline{RD} and \overline{WR} to get \overline{MEMR} and \overline{MEMW} signals. When both \overline{RD} and $\overline{IO/\overline{M}}$ signals go low, \overline{MEMR} signal goes low. Similarly, when both \overline{WR} and $\overline{IO/\overline{M}}$ signals go low, \overline{MEMW} signal goes low. To generate \overline{IOR} and \overline{IOW} signals for I/O operation, $\overline{IO/\overline{M}}$ signal is first inverted and then logically ORed with \overline{RD} and \overline{WR} signals.

| $\overline{IO/\overline{M}}$ | \overline{RD} | \overline{WR} | \overline{MEMR} $\overline{RD + IO/\overline{M}}$ | \overline{MEMW} $\overline{WR + IO/\overline{M}}$ | \overline{IOR} $\overline{RD + \overline{IO/\overline{M}}}$ | \overline{IOW} $\overline{WR + \overline{IO/\overline{M}}}$ |
|------------------------------|-----------------|-----------------|--|--|--|--|
| 0 | 0 | 0 | Condition never exists, because \overline{RD} and \overline{WR} signals does not go low simultaneously | | | |
| 0 | 0 | 1 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 |

| | | | | | | |
|---|---|---|--|---|---|---|
| 1 | 0 | 0 | Condition never exists, because \overline{RD} and \overline{WR} signals does not go low simultaneously | | | |
| 1 | 0 | 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Table 2.3

Same truth table can be implemented using 3:8 decoder as shown in Fig. 2.16.

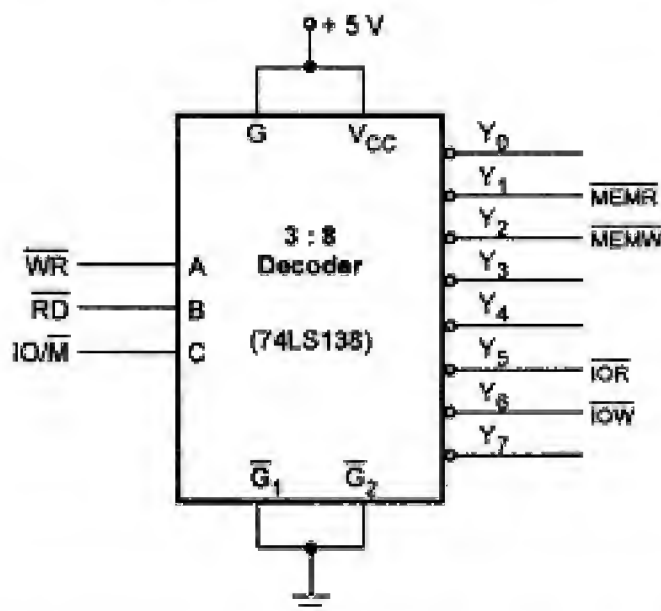


Fig. 2.16 Generation of control signals using 3:8 decoder

2.8.3 Bus Drivers

Typically, the 8085 buses can source 400 μ A and sink 2 mA of current, i.e. it can drive only one TTL load. Therefore, it is necessary to increase driving capacity of the 8085 buses. Bus drivers, buffers are used to increase the driving capacity of the buses.

Unidirectional Buffers :

As we know, the address bus is unidirectional, 8-bit unidirectional buffer, 74LS244 is used to buffer higher address bus. The Fig. 2.17 shows the logic diagram of 74LS244. It consists of eight non-inverting buffers with tri-state outputs. Each one can sink 24 mA and source 15 mA of current. These buffers are divided into two groups. The enabling and disabling of these groups are controlled by $\overline{1G}$ and $\overline{2G}$ lines.

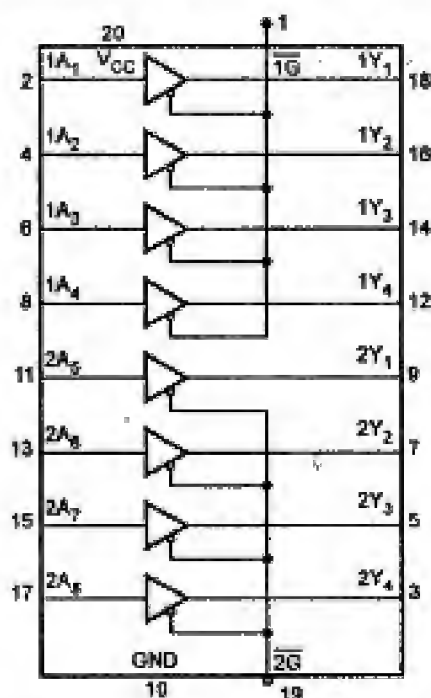
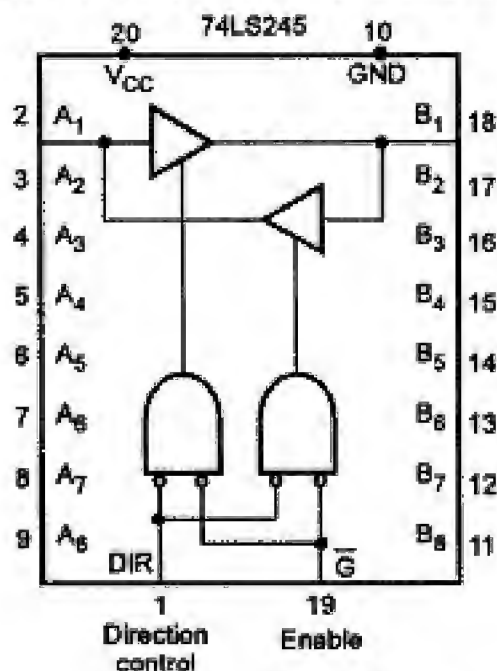


Fig. 2.17 Logic diagram of the 74LS244

Bi-directional Buffer :

To increase the driving capacity of data bus, bi-directional buffer is used. Fig. 2.18 shows the logic diagram of the bi-directional buffer 74LS245, also called an octal bus transceivers. It consists of sixteen non-inverting buffers, eight for each direction, with tri-state output. The direction of data flow is controlled by the pin DIR. When DIR is high, data flows from the A bus to the B bus; when it is low, data flows from B to A. The active low enable signal and the DIR signal are ANDed to activate the bus lines. Each buffer in this device can sink 24 mA and source 15 mA of current.



Function table

| Enable G | Direction control DIR | Operation |
|-------------|-----------------------------|-----------------|
| L | L | B Data to A Bus |
| L | H | A Data to B Bus |
| H | X | Isolation |

H=High level, L=Low level, X=Irrelevant

Fig. 2.18 Logic diagram of the 74LS245

2.8.4 Typical Configuration

Fig. 2.19 shows schematic of the 8085 microprocessor demultiplexed address bus and control signals.

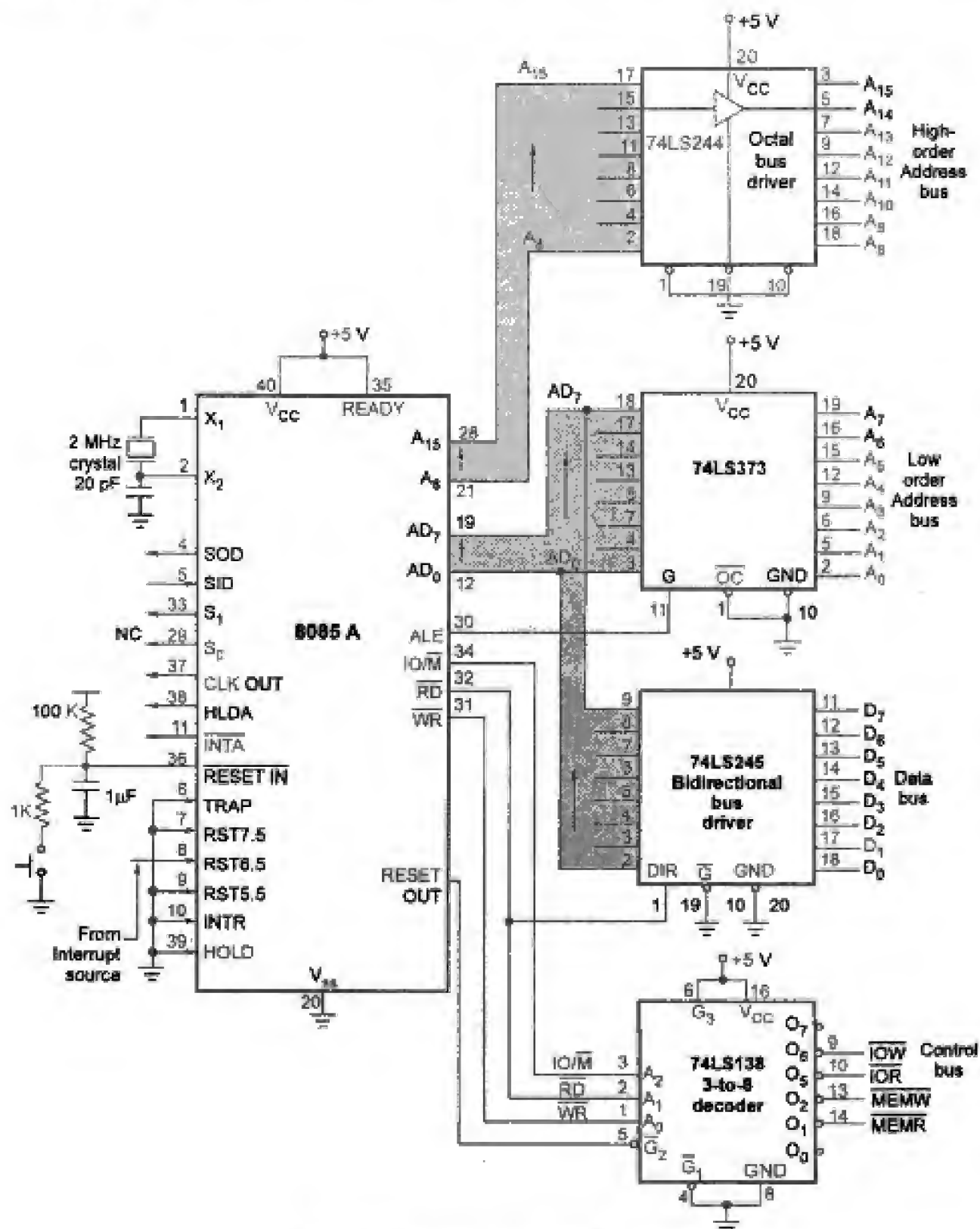


Fig. 2.19 Typical 8085 configuration

It also shows clock and reset circuits. Interrupt lines which are not in use are grounded. This is necessary because floating interrupt line may cause false triggering of interrupt. Similarly, since the DMA controller is not used, HOLD line is also grounded. As we know READY signal is used to synchronize slow peripherals with the microprocessor. When it is low, microprocessor enters in the wait state and when it is high, it indicates that the memory or peripheral is ready to send or receive data. Here, the READY signal is tied high to prevent the microprocessor from entering the wait state. ALE signal is connected to the clock input of the latch, to latch the low order address in T_1 of the machine cycle. To control the direction of the bi-directional buffer 74LS245, \overline{RD} signal from 8085 is connected to DIR input of the bi-directional buffer. Thus, when \overline{RD} signal is low, DIR is low and data flows from memory or I/O device to the microprocessor, performing read operation. When \overline{RD} signal is high, DIR is high and data flows from microprocessor to memory or I/O device performing write operation.

2.9 Instruction Cycle

During normal operation, the microprocessor sequentially fetches, decodes and executes one instruction after another until a halt instruction (HALT) is executed. When HALT instruction is executed, microprocessor enters in HALT state and remains in that state until reset signal is not activated. Upon activation of reset signal microprocessor again starts the fetch, decode and execute cycle, i.e. instruction cycle. This is illustrated in Fig. 2.20.

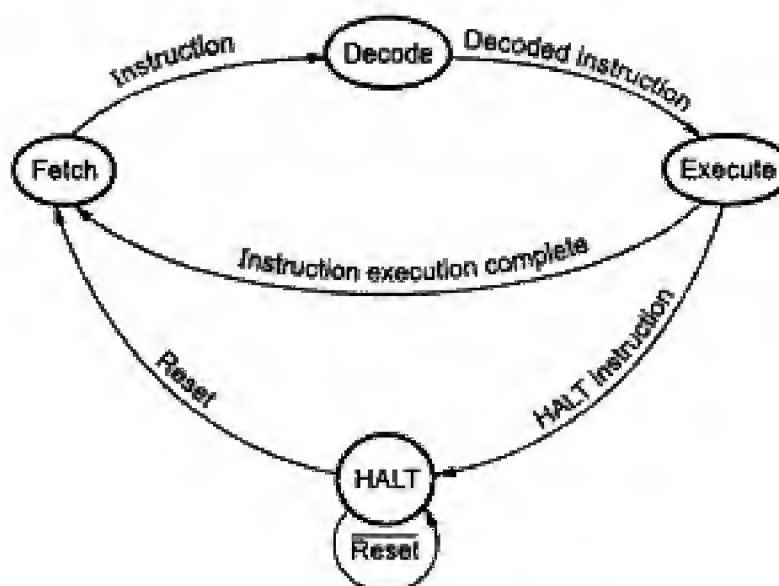


Fig. 2.20 Fetch, decode, execute cycle

The fetching, decoding and execution of a single instruction constitutes an **instruction cycle**, which consists of one to five read or write operations between processor and memory or input/output devices. Each memory or I/O operation requires a particular time period, called **machine cycle**. In other words, to move byte of data in or out of the microprocessor, a machine cycle is required. Each machine cycle consists of 3 to 6 clock periods/cycles, referred to as **T-states**. Therefore we can say that, one instruction cycle consists of one to five machine cycles and one machine cycle consists of three to six T-states i.e. three to six clock periods, as shown in the Fig. 2.21.

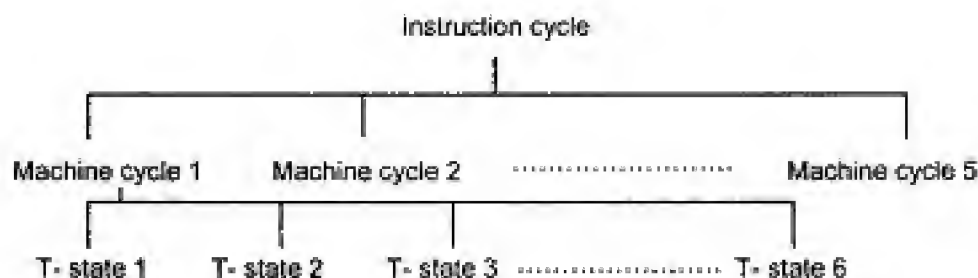


Fig. 2.21 Relation between instruction cycle, machine cycle and T-state

There are seven different types of machine cycles in the 8085A. Three status signals $\overline{IO/\overline{M}}$, S_1 and S_0 identify each type as shown in Table 2.4. These signals are generated at the beginning of each machine cycle and remain valid for the duration of the cycle.

| Machine Cycle | Status | | | Control | | |
|------------------|------------------------------|-------|-------|-----------------|-----------------|-------------------|
| | $\overline{IO/\overline{M}}$ | S_1 | S_0 | \overline{RD} | \overline{WR} | \overline{INTA} |
| Opcode Fetch | 0 | 1 | 1 | 0 | 1 | 1 |
| Memory Read | 0 | 1 | 0 | 0 | 1 | 1 |
| Memory Write | 0 | 0 | 1 | 1 | 0 | 1 |
| I/O Read | 1 | 1 | 0 | 0 | 1 | 1 |
| I/O Write | 1 | 0 | 1 | 1 | 0 | 1 |
| INTR Acknowledge | 1 | 1 | 1 | 1 | 1 | 0 |
| Bus Idle | 0 | 0 | 0 | 1 | 1 | 1 |

Table 2.4 8085 machine cycles

2.10 Instruction, Execution, Sequence and Data Flow

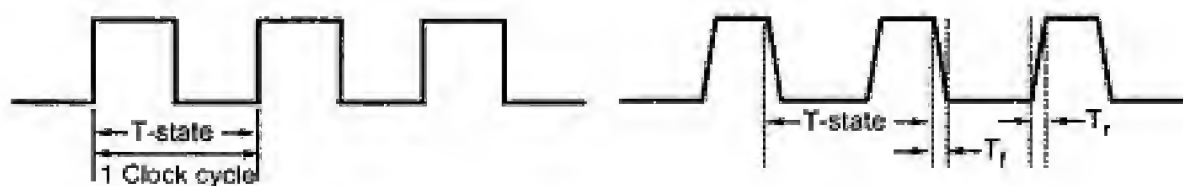
To execute an instruction, microprocessor has to perform sequence of machine cycles, depending on the instruction. In this section, we study how to represent different signals, their timings and sequence of machine cycles and data flow involved in it.

2.10.1 Representation of Signals

Before going to see the timing diagram, we will see the signals and its representation used in the timing diagrams.

Clock Signal :

The 8085 divides the clock frequency provided at x_1 and x_2 inputs by 2, which is called operating frequency. All the operations within the 8085 are synchronized with this operating frequency. Therefore in the timing diagram operating frequency clock is shown on the top and then the signals are shown with reference to operating frequency clock. Ideally, the clock signal should be square wave with zero rise time and fall time, as shown in the figure. But in practice, we don't get zero rise time and fall time. Therefore the clock and other signals are always shown with finite rise and fall times. Fig. 2.22 shows the practical way of representing clock signal.



(a) Ideal

(b) Practical

Fig. 2.22 Clock signal representation

Single Signal :

Single signal is represented by a line. It may have status either logic 0 or logic 1 or tri-state. The change in the state of the signal takes finite time and hence the state change of signal is represented with finite rise time and fall time, as shown in the Fig. 2.23.

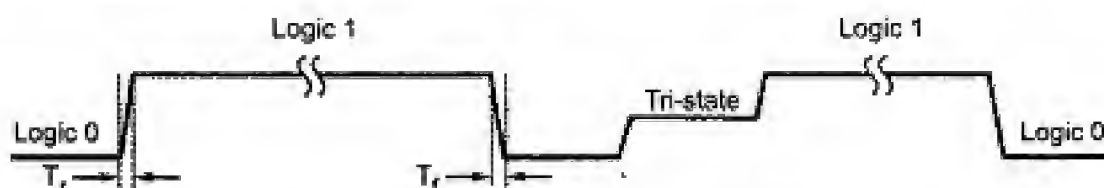


Fig. 2.23 Single signal representation

Group of Signals :

Group of signals is also called a bus e.g. address bus and data bus. To avoid complications in the timing diagram these signals are grouped and shown in the form of block as shown in Fig. 2.24.

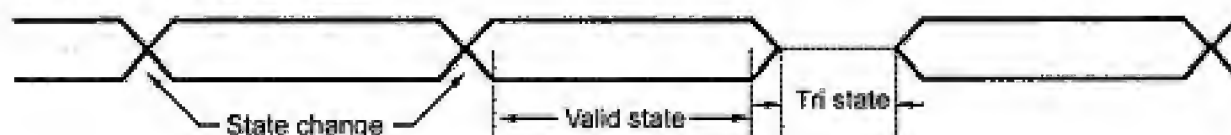


Fig. 2.24 Group of signals representation

In the group representation individual state is not considered, but the group state is considered. Change in state of single signal changes the state of group. It is represented by the cross as shown the Fig. 2.24. The tri-state condition of the group signals is shown by dotted lines. Two straight lines represent valid state/stable state.

In microprocessor systems, activation of signal/signals depends on the state of other signal/signals. Such situations are shown in the timing diagrams with the help of specific symbols. There are four possibilities :

Activation of a signal with the change in state of other signal.

Activation of a signal with the change in state of other signals.

Activation of signals with the change in state of other signal.

Activation of signals with the change in state of other signals.

Fig. 2.25 shows the representation of dependence of the signal/signals, in the timing diagram.

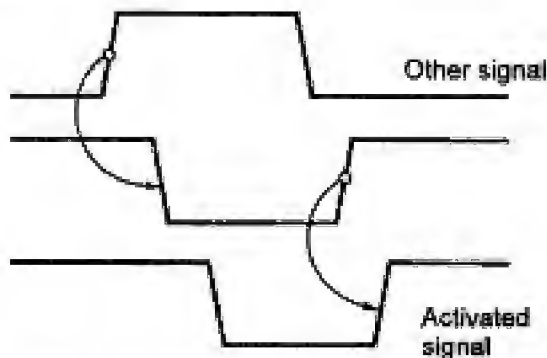


Fig. 2.25 a) Activation of signal with the change in state of other signal

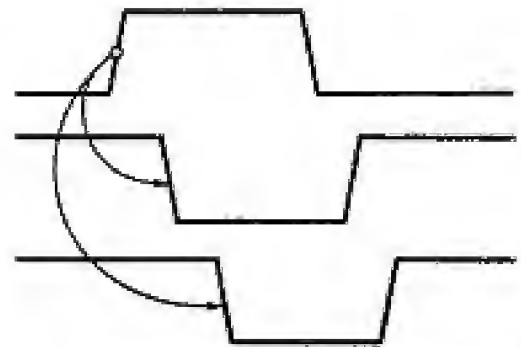


Fig. 2.25 b) Activation of signal with the change in state of other signal

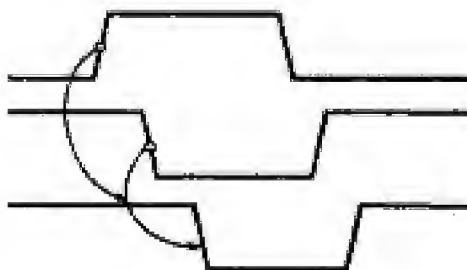


Fig. 2.25 c) Activation of a signal with the change in state of other signals

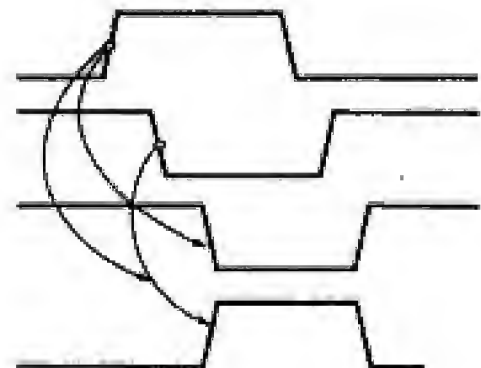


Fig. 2.25 d) Activation of signals with the change in state of other signals

2.10.2 Signal Timings

In 8085 microprocessor, signals are activated at specific instant for specific time period. Once we understand this, it is very easy to draw timing diagrams. The following section explains when the signals are activated and for what period they remain in active state.

ALE (Address Latch Enable) :

This signal is active high signal. It is activated in the beginning of the T_1 state of each machine cycle, except bus idle machine cycle, and it remains active in the T_1 state as shown in the Fig. 2.26.

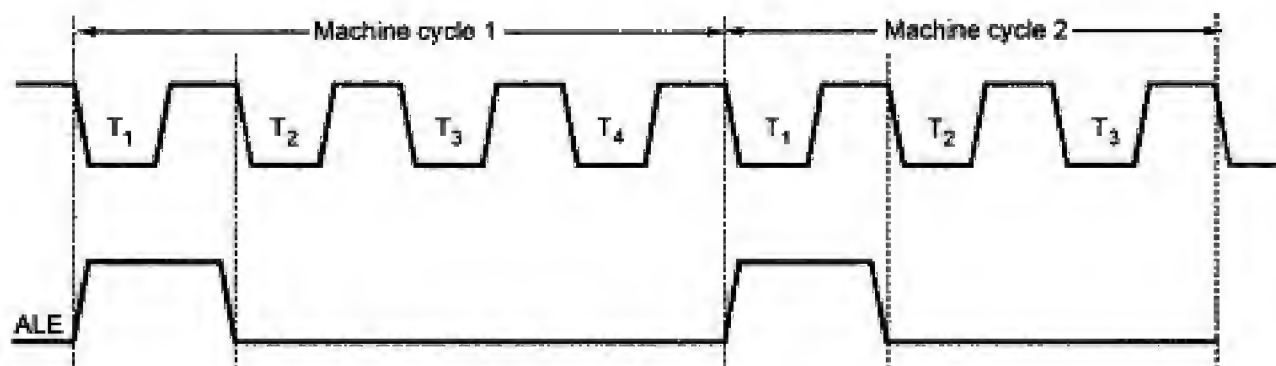


Fig. 2.26 ALE activation and its period

$A_0 - A_7$ (Lower byte address) :

The lower byte of address is available on the multiplexed address/data bus ($AD_0 - AD_7$) during T_1 state of each machine cycle, except bus idle machine cycle, as shown in Fig. 2.27.

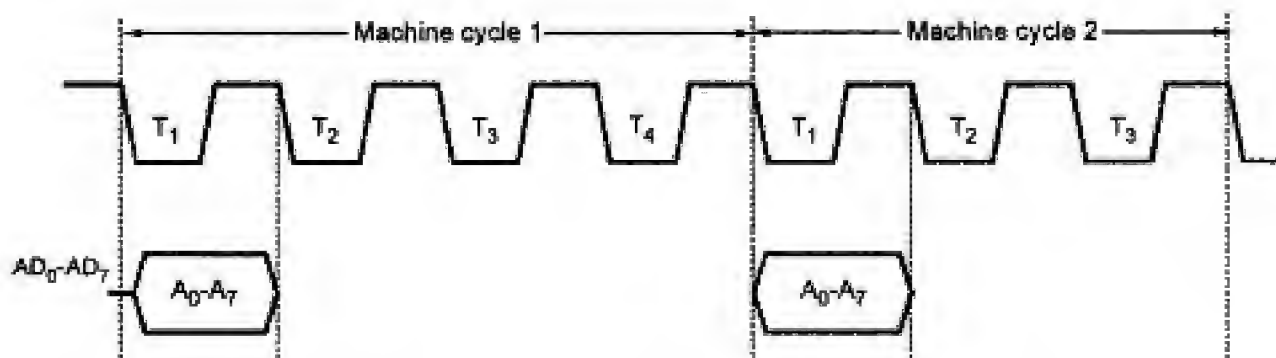
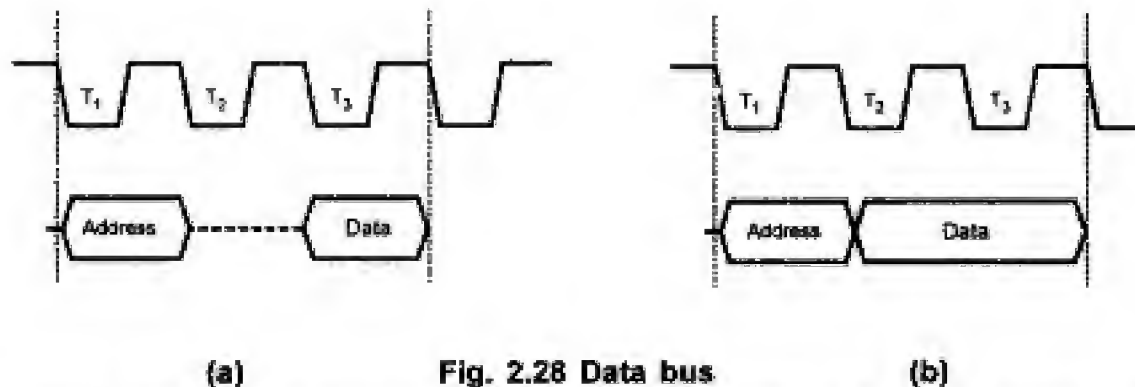


Fig. 2.27 Lower address on the multiplexed bus

$D_0 - D_7$ (Data bus) :

The data from memory or I/O device and from microprocessor to memory or I/O device is transferred during T_2 and T_3 - states. It is important to note that in read machine cycle, data will appear on the data bus during the later part of the T_2 - state, as shown in the Fig. 2.28, whereas in write cycle data will appear on the data bus at the beginning of the T_2 - state, as shown in the Fig. 2.28.



(a)

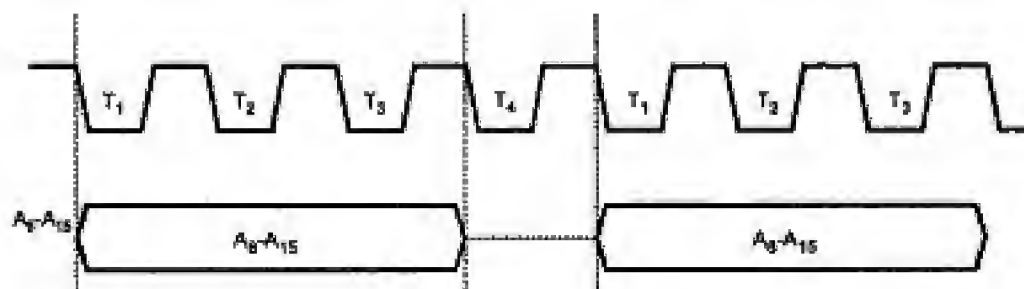
Fig. 2.28 Data bus

(b)

To read data from memory or I/O device it is necessary to select memory or I/O device. After selection, device will put the data from selected location on the data bus. This action needs finite time. This time is referred to as 'access time'. In case of write cycle, data is available in the register set of the microprocessor and it can put that data on the data bus with zero access time.

 $A_8 - A_{15}$ (Higher byte address) :

The higher byte of address is available on the $A_8 - A_{15}$ bus during T_1 , T_2 and T_3 - state of each machine cycle, except bus idle machine cycle, as shown in Fig. 2.29.

Fig. 2.29 Higher byte address on $A_8 - A_{15}$

$\overline{IO/\overline{M}}$, S_0 , S_1 :

These signals are called status signals. They decide the type of machine cycle to be executed. They are activated at the beginning of T_1 -state of each machine cycle and remain active till the end of the machine cycle.

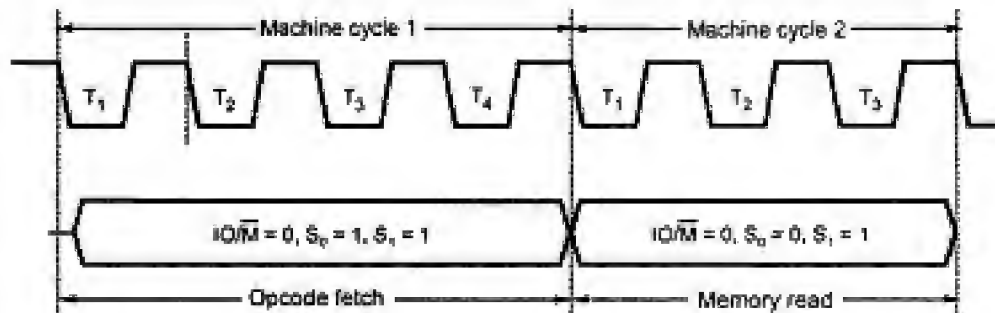


Fig. 2.30 Status signals

\overline{RD} and \overline{WR} : These signals decide the direction of the data transfer. When \overline{RD} signal is active, data is transmitted from memory or I/O device to the microprocessor, and when \overline{WR} signal is active, data is transmitted from microprocessor to the memory or I/O device. Both signals are never active at a time.

As we know data transfer in 8085 takes place during T_2 and T_3 , these signals are activated during T_2 and T_3 , as shown in the Fig. 2.31.

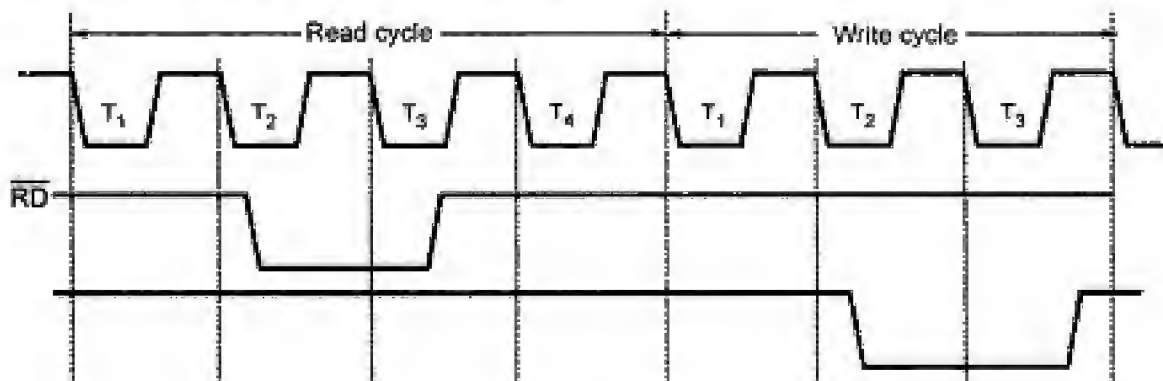


Fig. 2.31 \overline{RD} and \overline{WR} signals

2.10.3 Machine Cycles

The 8085 has seven machine cycles. These are :

1. Opcode Fetch
2. Memory Read
3. Memory Write
4. I/O Read

5. I/O Write
6. Interrupt Acknowledge
7. Bus Idle

In each machine cycles, particular signals are activated and data flow takes place in terms of opcode or operand to carry out instruction execution. Let us see which signals are activated and how data flow takes place in each machine cycle one by one.

1. Opcode Fetch Cycle :

The first machine cycle of every instruction is opcode fetch cycle in which the 8085 finds the nature of the instruction to be executed. In this machine cycle, processor places the contents of the program counter on the address lines, and through the read process, reads the opcode of the instruction. Fig. 2.32 (a) shows flow of data (opcode) from memory to the microprocessor and Fig. 2.32 (b) shows the timing diagram for opcode fetch machine cycle. The length of this cycle is not fixed. It varies from 4T states to 6T states as per the instruction. The following section describes the opcode fetch cycle in step by step manner.

Step 1 : (State T_1) In T_1 state, the 8085 places the contents of program counter on the address bus. The high-order byte of the PC is placed on the A_8-A_{15} lines. The low-order byte of the PC is placed on the $AD_0 - AD_7$ lines which stays on only during T_1 . Thus microprocessor activates ALE (Address Latch Enable) which is used to latch the low-order byte of the address in external latch before it disappears.

In T_1 , 8085 also sends status signals IO/\overline{M} , S_1 and S_0 . IO/\overline{M} specifies whether it is a memory or I/O operation, S_1 status specifies whether it is read/write operation; S_1 and S_0 together indicate read, write, opcode fetch, machine cycle operation, or whether it is in HALT state. In opcode fetch machine cycle status signals are : $IO/\overline{M} = 0$, $S_1 = 1$, $S_0 = 1$.

Step 2 : (State T_2) In T_2 , low-order address disappears from the $AD_0 - AD_7$ lines. (However $A_0 - A_7$ remain available as they were latched during T_1). In T_2 , 8085 sends \overline{RD} signal low to enable the addressed memory location. The memory device then places the contents of addressed memory location on the data bus ($AD_0 - AD_7$).

Step 3 : (State T_3) During T_3 , 8085 loads the data from the data bus in its Instruction Register and raises \overline{RD} to high which disables the memory device.

Step 4 : (State T_4)

In T_4 , microprocessor decodes the opcode, and on the basis of the instruction received, it decides whether to enter state T_5 or to enter state T_1 of the next machine cycle. One byte instructions those operate on eight bit data (8 bit operand) are executed in T_4 .

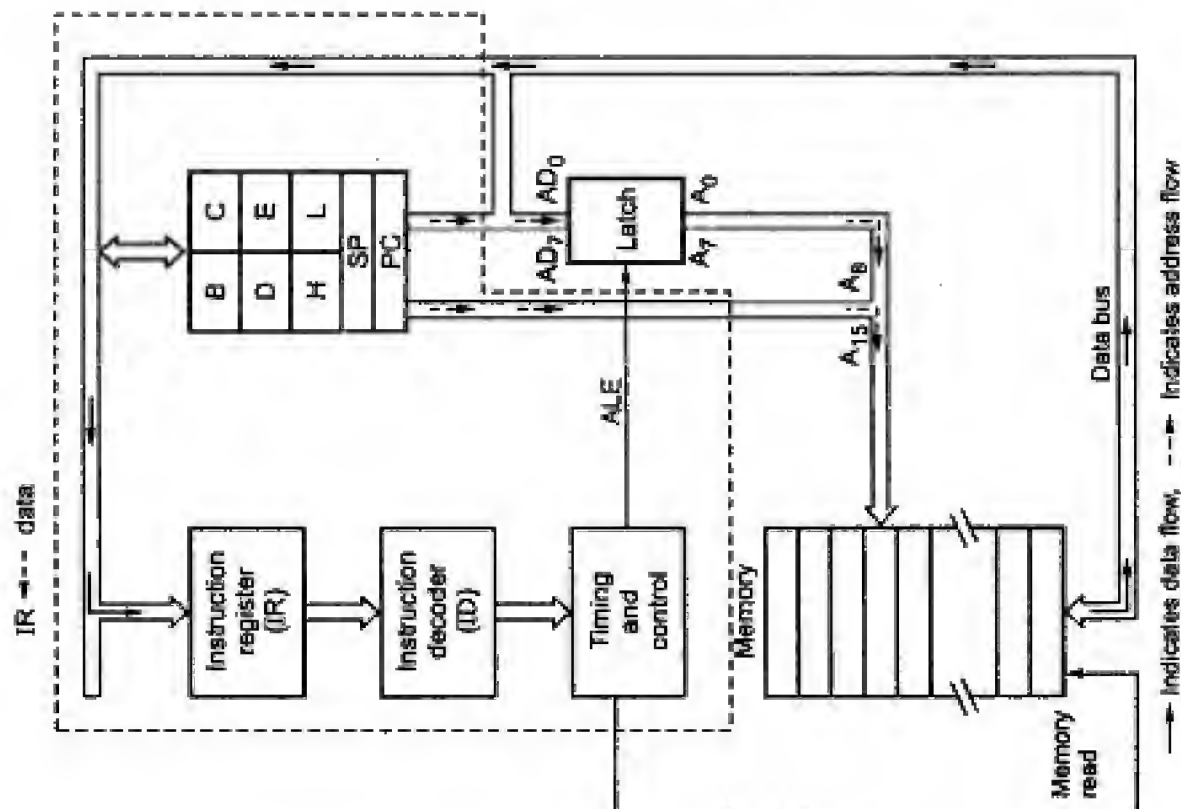


Fig. 2.32 (a) Data (opcode) flow from memory to microprocessor

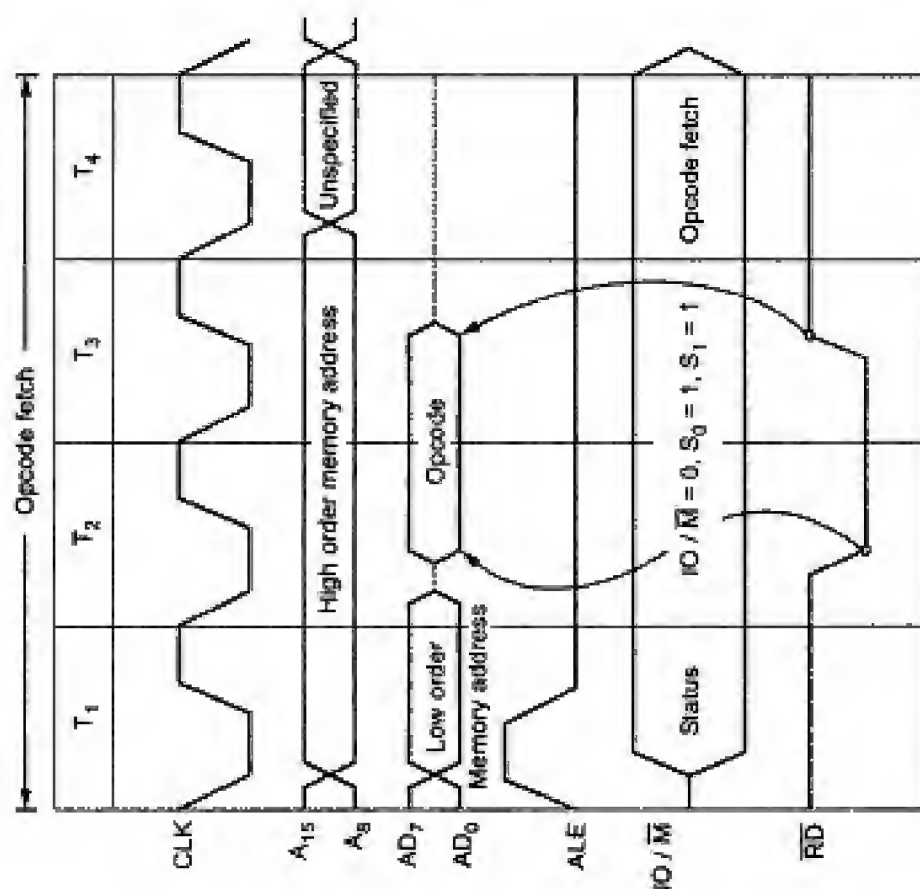


Fig. 2.32 (b) Opcode fetch machine cycle

For example : MOV A,B, ANA D, ADD B, INR L, DCR C, RAL and many more.

Note : For one byte instructions which operate on eight bit data, data is always available in the internal memory of 8085 i.e. registers.

Step 5 : (State T_5 and T_6) States T_5 and T_6 , when entered, are used for internal microprocessor operations required by the instruction. During T_5 and T_6 , 8085 performs stack write, internal 16 bit, and conditional return operations depending upon the type of instruction. One byte instructions those operate on sixteen bit data (16 bit operand) are executed in T_5 and T_6 . For example DCX H, PCHL, SPHL, INX H, etc.

2. Memory Read Cycle :

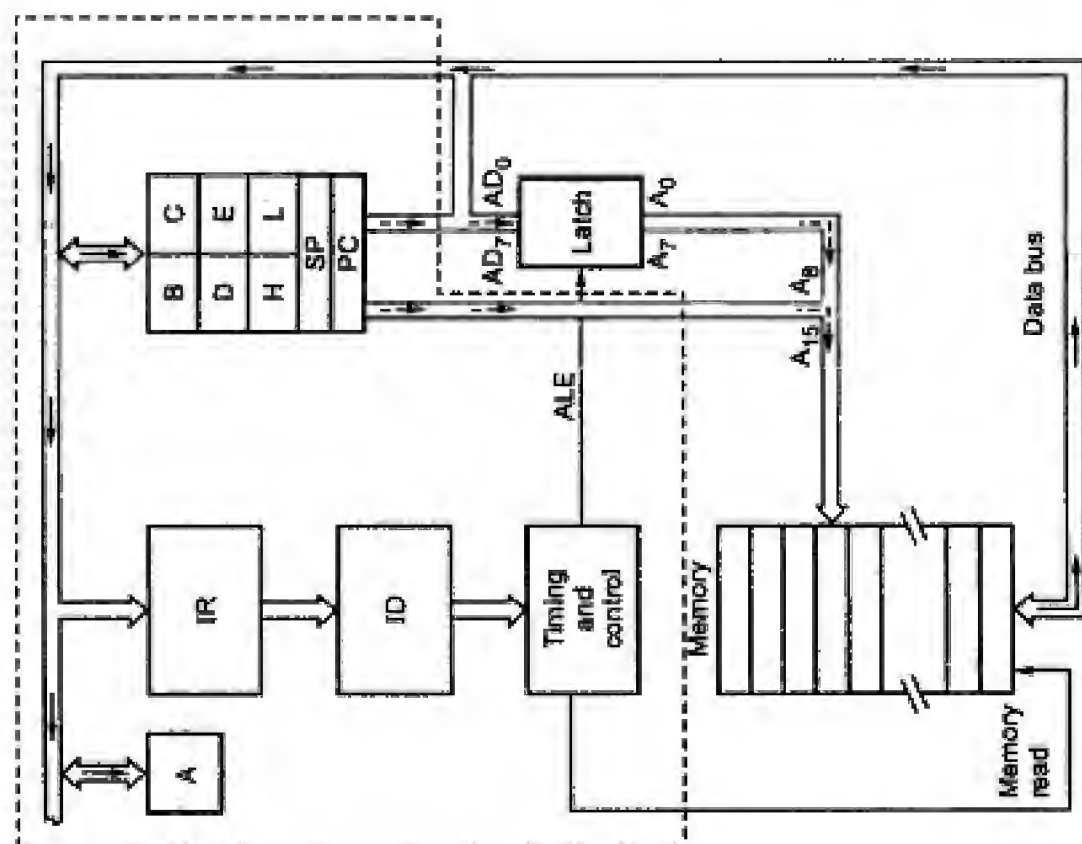
The 8085 executes the memory read cycle to read the contents of R/W memory or ROM. The length of this machine cycle is 3-T states ($T_1 - T_3$). In this machine cycle, processor places the address on the address lines from the stack pointer, general purpose register pair or program counter, and through the read process, reads the data from the addressed memory location. Fig. 2.33 (a) shows flow of data from memory to the microprocessor and Fig. 2.33 (b) shows the timing diagram for memory read machine cycle. Memory read machine cycle is similar to the opcode fetch machine cycle. However, they use only states T_1 to T_3 , and the status signal values ($IO/\overline{M} = 0$, $S_1 = 1$, $S_0 = 0$) appropriate for memory read machine cycle are issued in T_1 . The following section describes the memory read machine cycle in step by step manner.

Step 1 : (State T_1) In T_1 state, microprocessor places the address on the address lines from stack pointer, general purpose register pair or program counter and activates ALE signal in order to latch low-order byte of address.

During T_1 , 8085 sends status signals : $IO/\overline{M} = 0$, $S_1 = 1$, and $S_0 = 0$ for memory read machine cycle.

Step 2 : (State T_2) In T_2 , 8085 sends \overline{RD} signal low to enable the addressed memory location. The memory device then places the contents of addressed memory location on the data bus ($AD_0 - AD_7$).

Step 3 : (State T_3) During T_3 , 8085 loads the data from the data bus into specified register (F, A, B, C, D, E, H, and L) and raises \overline{RD} to high which disables the memory device.



— Indicates data flow, - - - Indicates address flow

Fig.2.33 (a) Data flow from memory to microprocessor

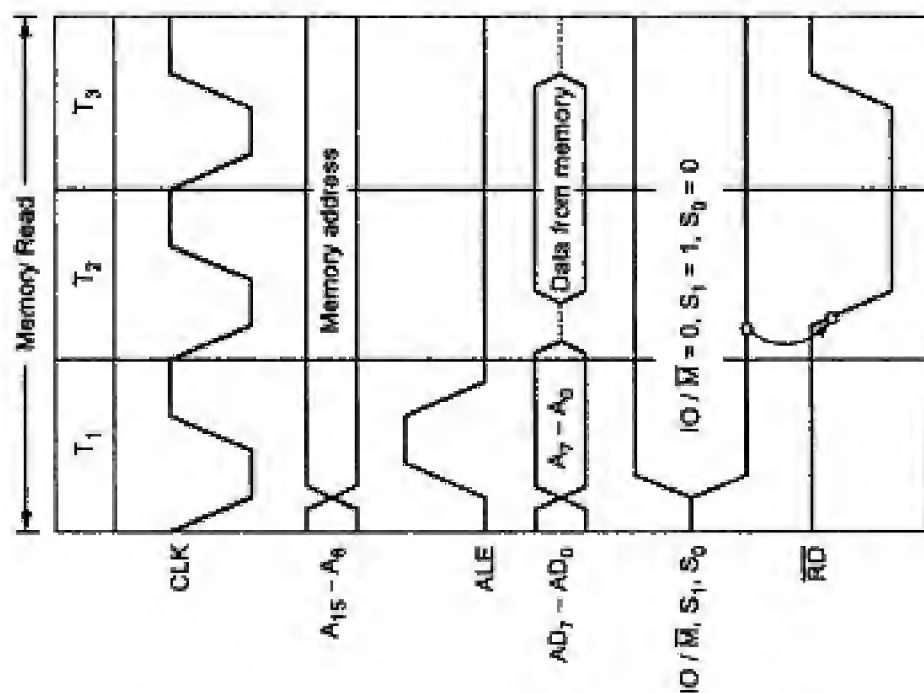


Fig. 2.33 (b) Memory read machine cycle

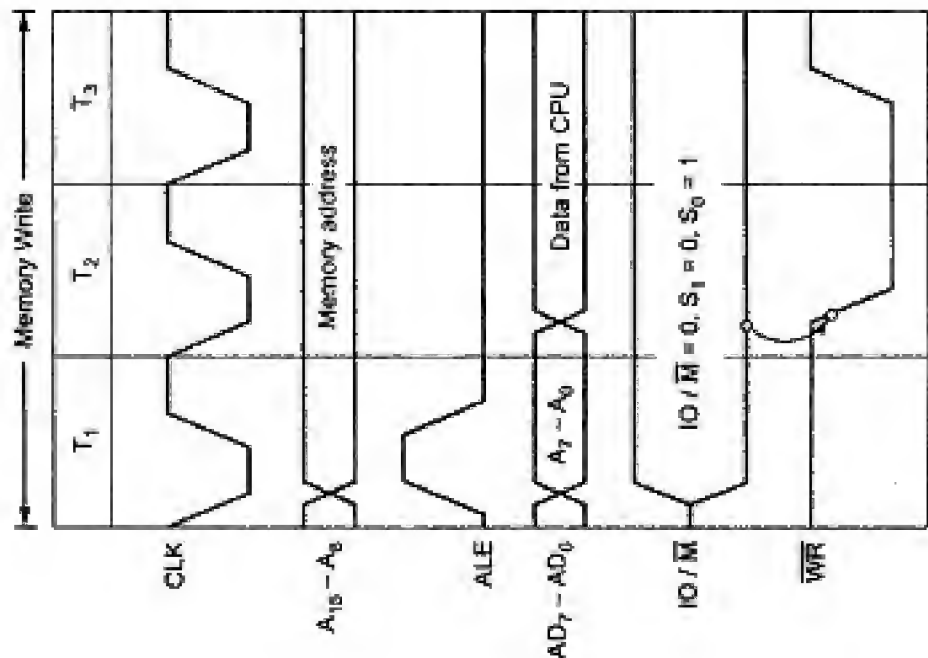


Fig. 2.34 (b) Memory write machine cycle

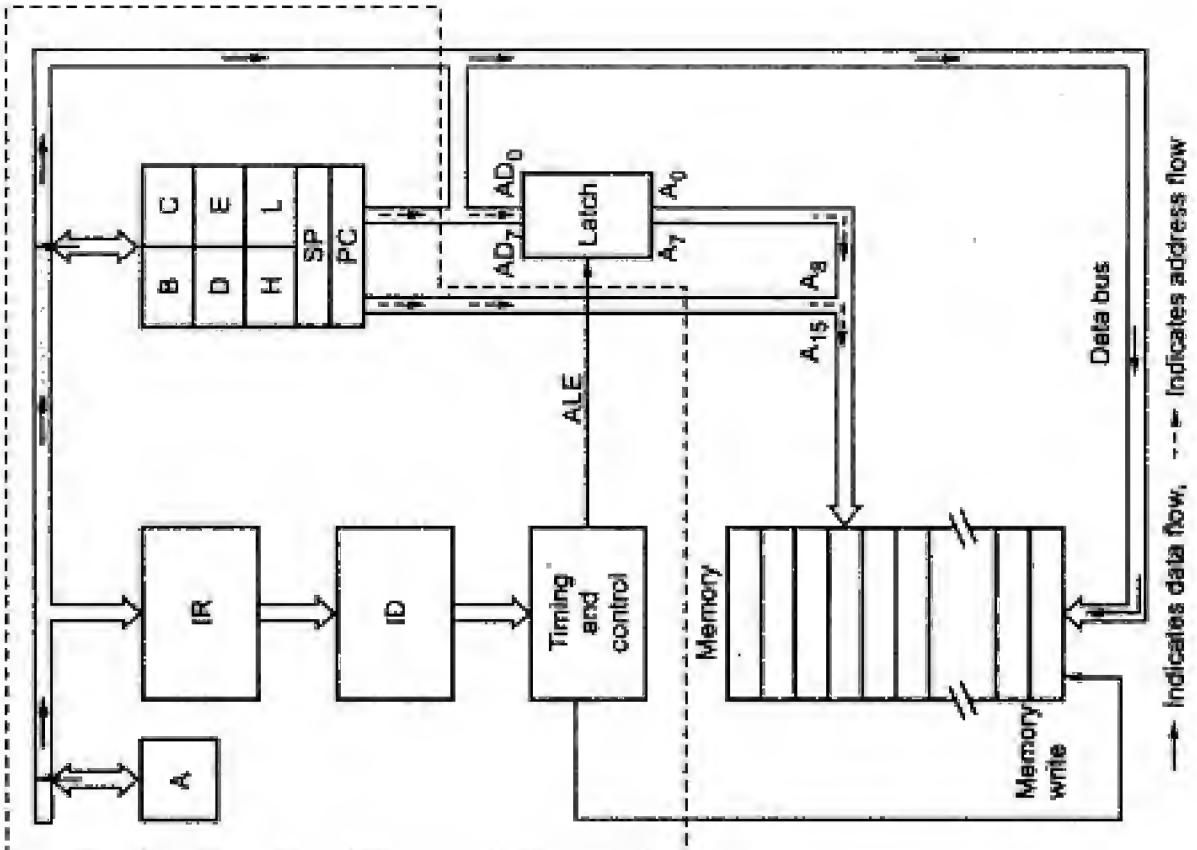


Fig. 2.34 (a) Data flow microprocessor to memory

3. Memory Write Cycle

The 8085 executes the memory write cycle to store the data into data memory or stack memory. The length of this machine cycle is 3T states ($T_1 - T_3$). In this machine cycle, processor places the address on the address lines from the stack pointer or general purpose register pair and through the write process, stores the data into the addressed memory location. Fig. 2.34 shows the timing diagram for memory write machine cycle. The memory write timing diagram is similar to the memory read timing diagram, except that instead of \overline{RD} , \overline{WR} signal goes low during T_2 and T_3 . The status signals for memory write cycle are : $IO/\overline{M} = 0$, $S_1 = 0$, $S_0 = 1$. The following section describes the memory write machine cycle in step by step manner.

Step 1 : (State T_1) In T_1 state, the 8085 places the address on the address lines from stack pointer or general purpose register pair and activates ALE signal in order to latch low-order byte of address. During T_1 , 8085 sends status signals :

$IO/\overline{M} = 0$, $S_1 = 0$ and $S_0 = 1$ for memory write machine cycle.

Step 2 : (State T_2) In T_2 , 8085 places data on the data bus and sends \overline{WR} signal low for writing into the addressed memory location.

Step 3 : (State T_3) During T_3 , \overline{WR} signal goes high, which disables the memory device and terminates the write operation.

4, 5. I/O Read and I/O Write Cycles

The I/O read and I/O write machine cycles are similar to the memory read and memory write machine cycles, respectively, except that the IO/\overline{M} signal is high for I/O read and I/O write machine cycles. High IO/\overline{M} signal indicates that it is an I/O operation. Fig. 2.35 and Fig. 2.36 show the timing diagrams for I/O read and I/O write cycles, respectively.

6. Interrupt Acknowledge Cycle

In response to INTR signal, 8085 executes interrupt acknowledge machine cycle to read an instruction from the external device. Theoretically, the external device can place any instruction on the data bus in response to \overline{INTA} . However, only RST and CALL, save the PC contents (return address) before transferring control to the interrupt service routine. The next sections explain interrupt acknowledge cycles for RST and CALL instructions.

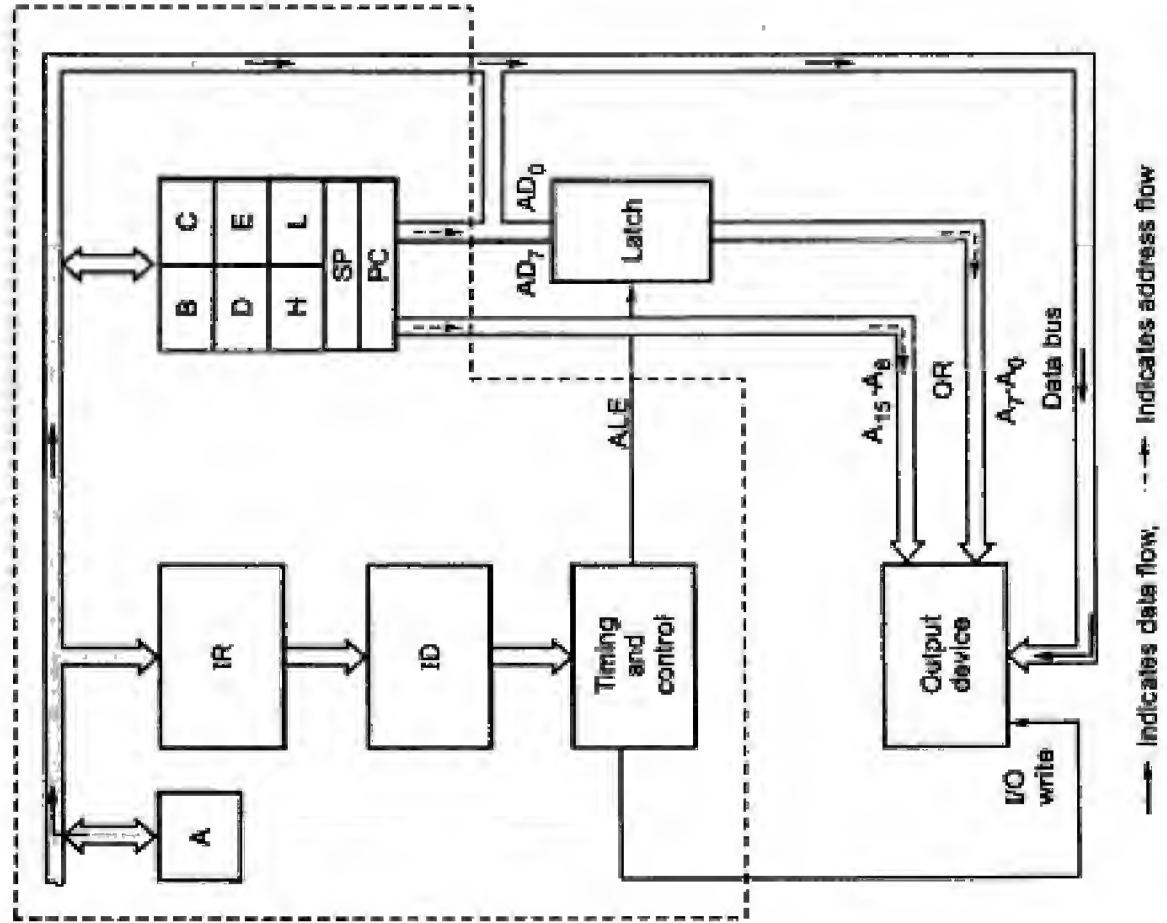


Fig. 2.36 (a) Data flow from microprocessor to output device

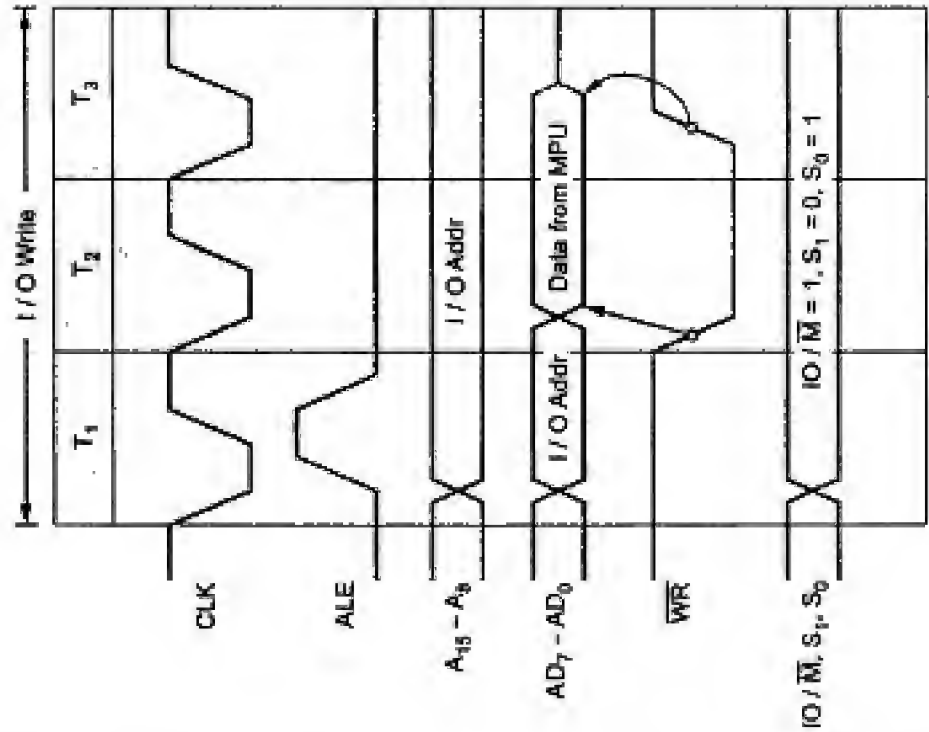


Fig. 2.36 (b) I/O write machine cycle

Interrupt acknowledge cycle for RST instruction

Fig. 2.37 shows the timing diagram of the interrupt acknowledge machine cycle and execution of RST instruction. The interrupt acknowledge cycle is similar to the opcode fetch cycle, with two exceptions.

1. The $\overline{\text{INTA}}$ signal is activated instead of the $\overline{\text{RD}}$ signal.
2. The status lines ($\text{IO}/\overline{\text{M}}$, S_0 and S_1) are 111 instead of 011.

During interrupt acknowledge machine cycle (M_1), the RST is decoded, which initiates 1 byte CALL instruction to the specific vector location. The machine cycles M_2 and M_3 are memory write cycles that store the contents of the program counter on the stack, and then a new instruction cycle begins.

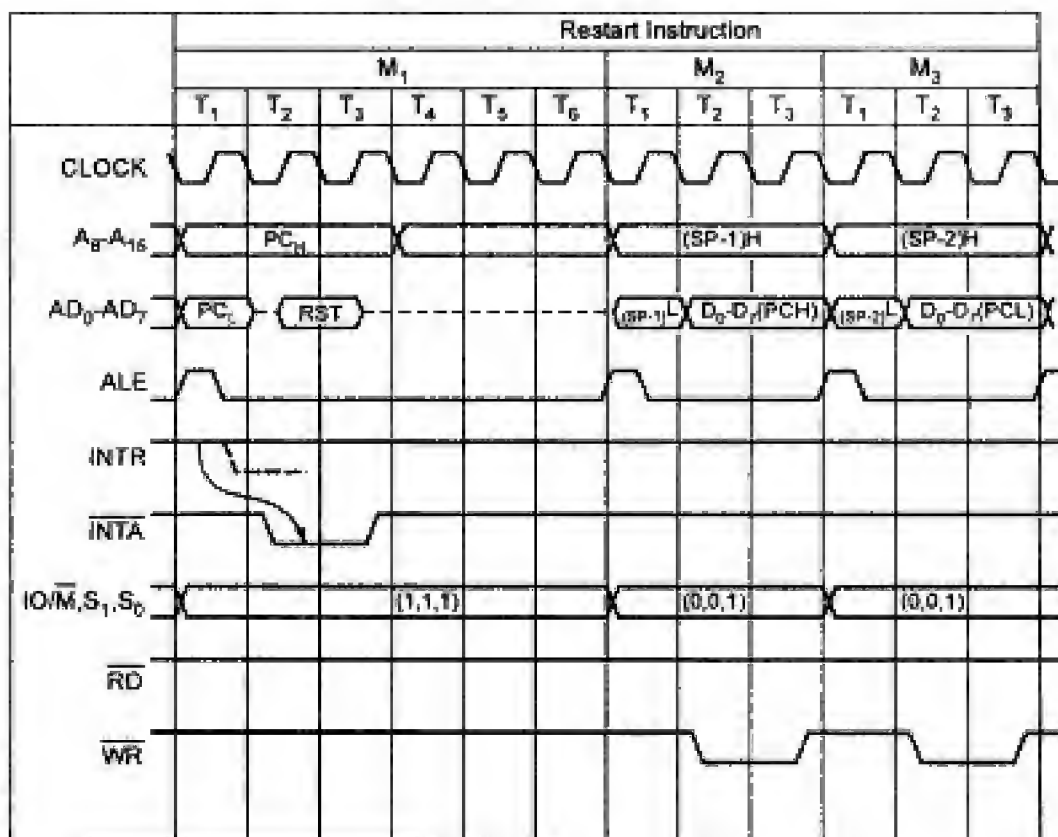
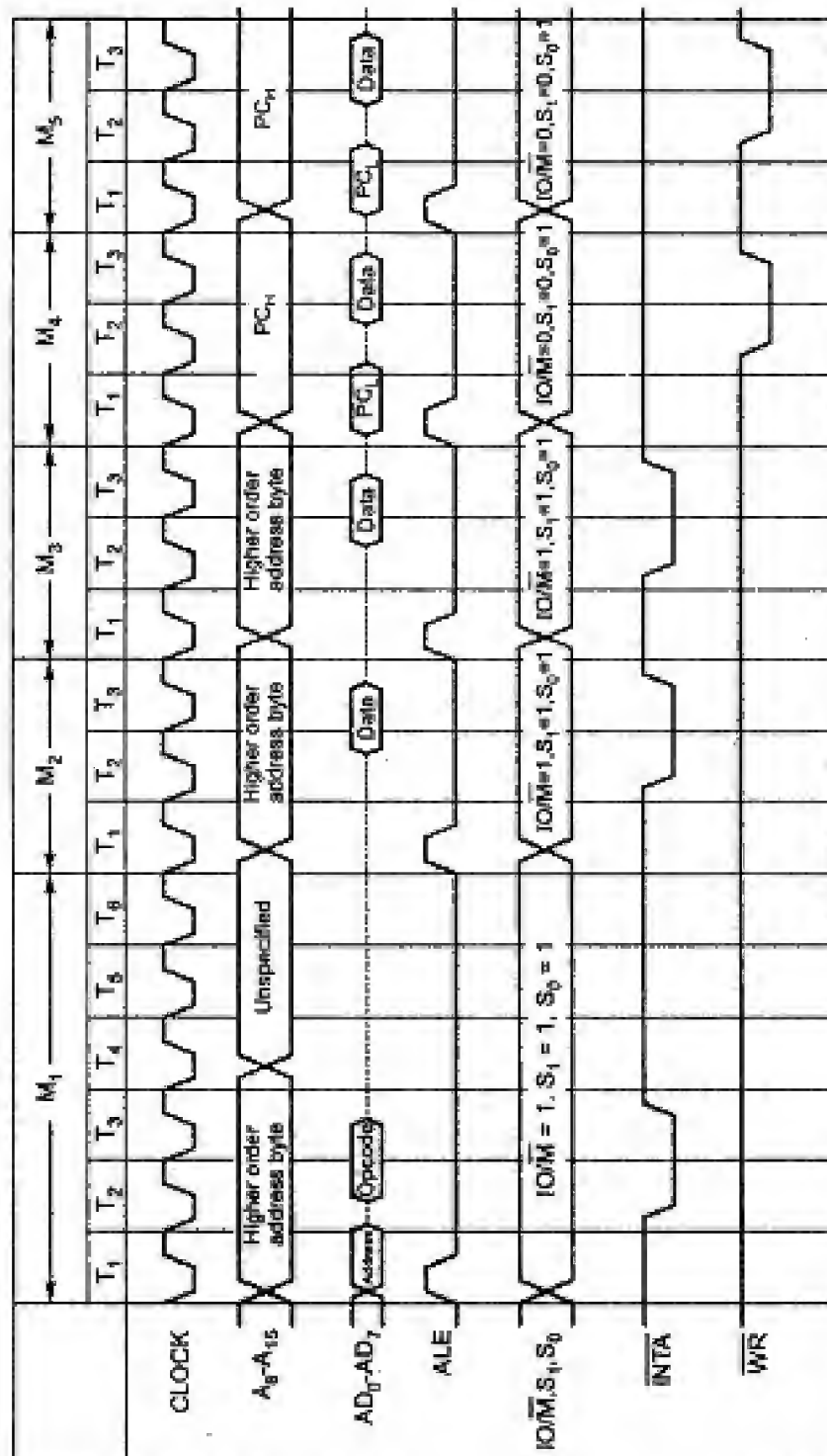


Fig. 2.37 Restart instruction

Interrupt acknowledge cycle for CALL instruction

Fig. 2.38 shows the timing diagram of the interrupt acknowledge machine cycle and execution of a CALL instruction. For CALL instruction, it is necessary to fetch the two bytes of the CALL address through two additional interrupt acknowledge machine cycles (M_2 and M_3 in Fig. 2.38). The machine cycles M_4 and M_5 are memory write cycles that store the contents of the program counter on the stack, and then a new instruction cycle begins.

Fig. 2.36 Timing diagram of $\overline{\text{INTA}}$ machine cycle and execution of call instruction

7. Bus Idle Cycle

There are few situations where the machine cycles are neither Read nor Write. These situations are :

1. For execution of DAD instruction (this instruction adds the contents of a specified register pair to the contents of HL register pair) ten T states are required. This means that

after execution of opcode fetch machine cycle, DAD instruction requires 6 extra T-states to add 16 bit contents of a specified register pair to the contents of HL register pair. These extra T-states which are divided into two machine cycles do not involve any memory or I/O operation. These machine cycles are called BUS IDLE machine cycles. Fig. 2.39 shows Bus Idle Machine Cycle for DAD instruction.

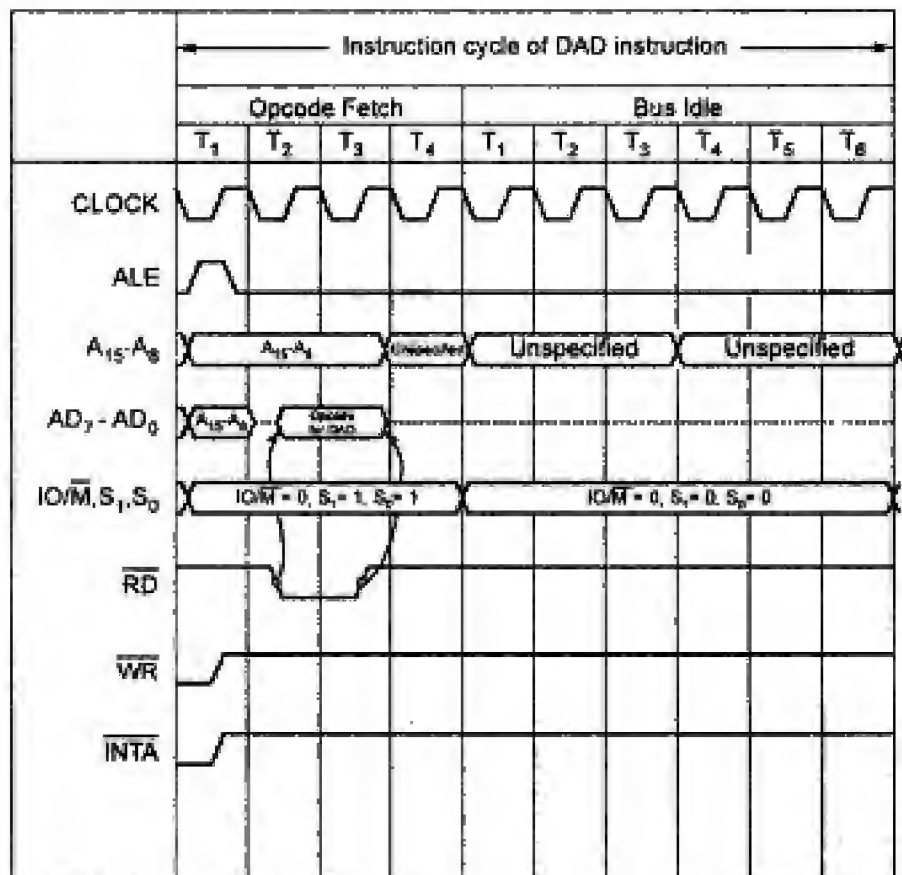


Fig. 2.39 Timing diagram for DAD instruction

In the case of DAD, these Bus Idle cycles are similar to memory read cycles, except $\overline{\text{RD}}$ and ALE signals are not activated.

2. During internal opcode generations, for TRAP and RST interrupts, 8085 executes Bus Idle Machine Cycles. Fig. 2.40 shows the Bus Idle Machine Cycle for TRAP. In response to TRAP interrupt, 8085 enters into a Bus Idle Machine Cycle during which it invokes restart instruction, stores the contents of PC onto the stack and places 0024H (Vector address of TRAP) onto the program counter.

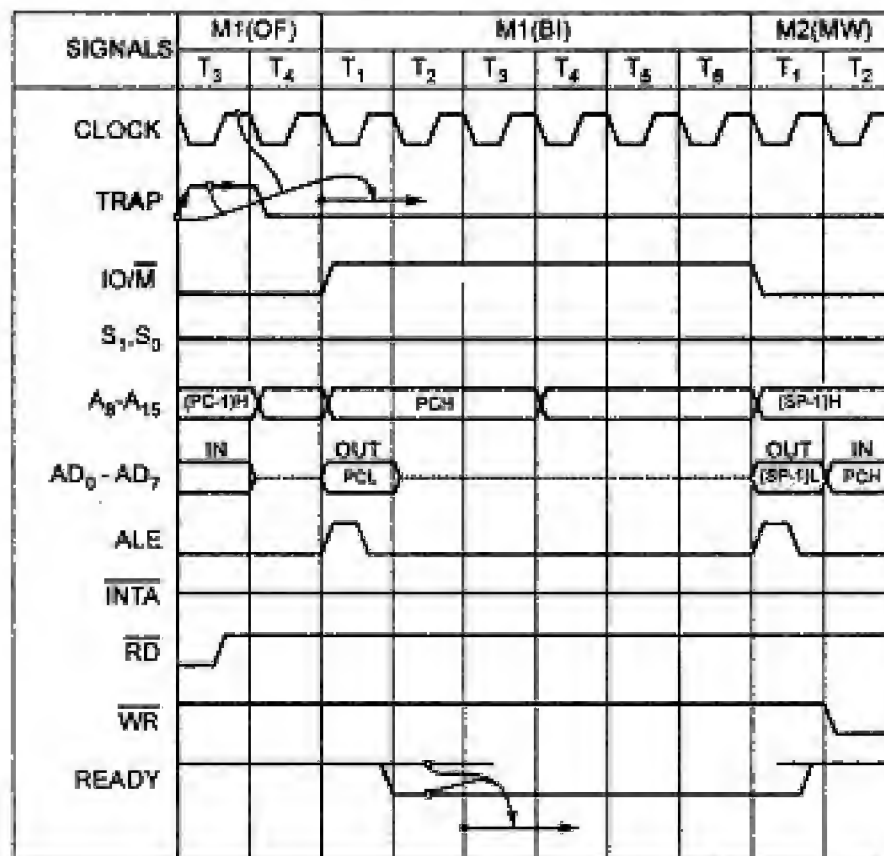


Fig. 2.40 Bus idle machine cycle for trap

The number of machine cycles required to fetch complete instruction depends on the instruction type :

1. One byte
2. Two byte
3. Three byte

One byte instruction doesn't require any additional machine cycle. Two byte instruction requires one additional memory read machine cycle, whereas three byte instruction requires two additional memory read machine cycles.

The number of machine cycles required to execute the instruction depends on the particular instruction. The total number of machine cycles required varies from one to five. It is possible that memory read and memory write machine cycles occur more than once in a single instruction cycle. The following examples illustrate the timing diagrams and machine cycles used for few 8085 instructions.

➡ **Example 2.1 :** Draw the timing diagram for instruction *MVI A, 30H* which is stored at address 2000H.

| | |
|-------|----|
| 2000H | 3E |
| 2001H | 30 |

Solution : This instruction consists of two bytes; the first is the opcode for MVI A instruction and the second is the data byte. The 8085 needs to read these bytes first from memory and thus requires at least two machine cycles. The first machine cycle is opcode fetch and second is memory read. Fig. 2.41 shows the timing diagram for this instruction. This instruction cycle is described in the following paragraphs.

1. The first machine cycle is an opcode fetch machine cycle. In T_1 , the microprocessor places the 16 bit memory address (2000H) from the program counter on the address bus, 20H on the $A_{15} - A_8$, and 00H on $AD_7 - AD_0$ and increments program counter to 2001H to point to the next memory location. It activates ALE signal (active high) to latch the low-order address 00H from the bus $AD_7 - AD_0$. It also gives the status signals (IO/\overline{M} , S_1 and S_0) 011 to indicate that it is an opcode fetch machine cycle. In T_2 , the 8085 activates \overline{RD} (active low) and reads the contents of memory location 2000H i.e. 3EH. Then 8085 places the opcode in the instruction register and disables the \overline{RD} signal during T_3 . During T_4 , the 8085 decodes the opcode and recognizes that it needs memory read machine cycle to read second byte of the instruction. In T_4 , the contents of the bus $A_{15} - A_8$ are unknown, and the data bus $AD_7 - AD_0$ goes into high impedance state.

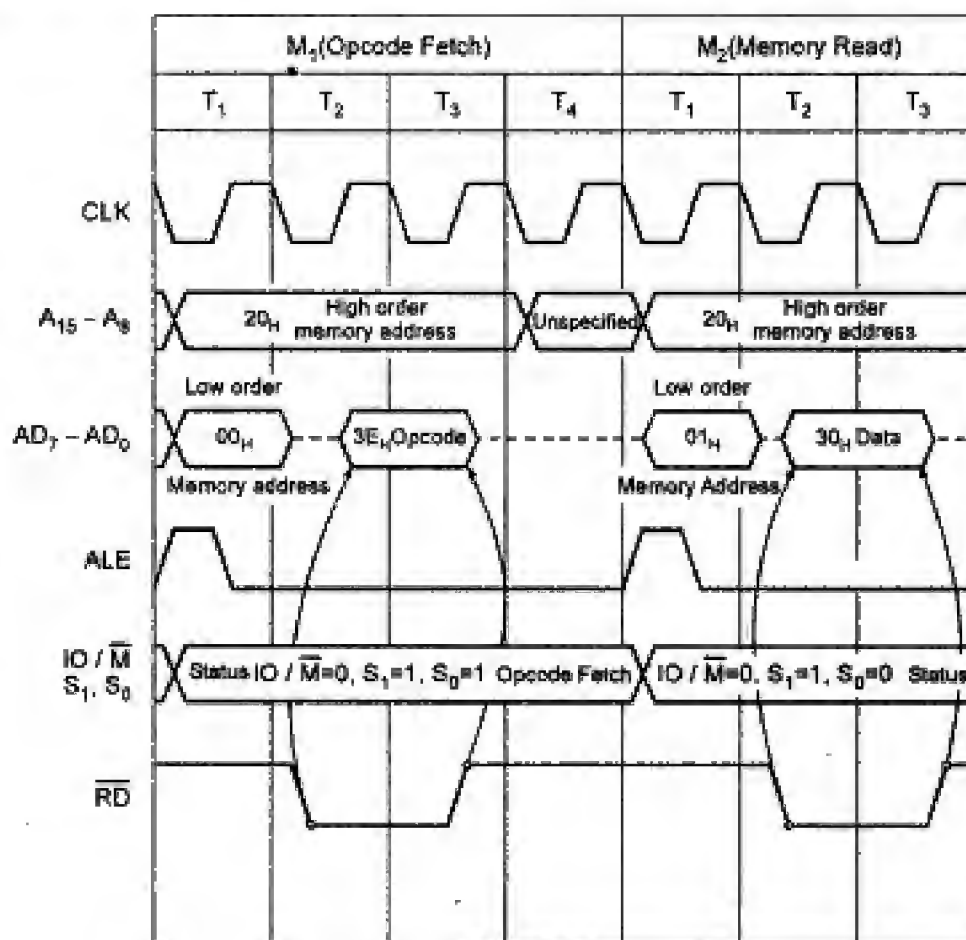


Fig. 2.41 Timing diagram for MVI A, 30H instruction

2. After opcode fetch machine cycle 8085 executes memory read machine cycle. In T_1 of memory read machine cycle, 8085 places the address 2001H on the address bus and increments the program counter to point the next memory location (2002H). 8085 then activates ALE signal and sends status signals ($\overline{IO/\overline{M}}$, S_1 and S_0) 010 to indicate memory read machine cycle. During T_2 and T_3 , 8085 activates \overline{RD} signal and reads the 8 bit data from memory location 2001H (30H). The 8085 then stores this data into the accumulator.

➡ **Example 2.2 :** Indicate machine cycles and T-States required for execution of STA instruction.

Solution : For instruction shown in Fig. 2.42, three machine cycles are required to fetch the instruction and one additional machine cycle is required to store the contents of accumulator into the memory. So following machine cycles are required for STA instruction.

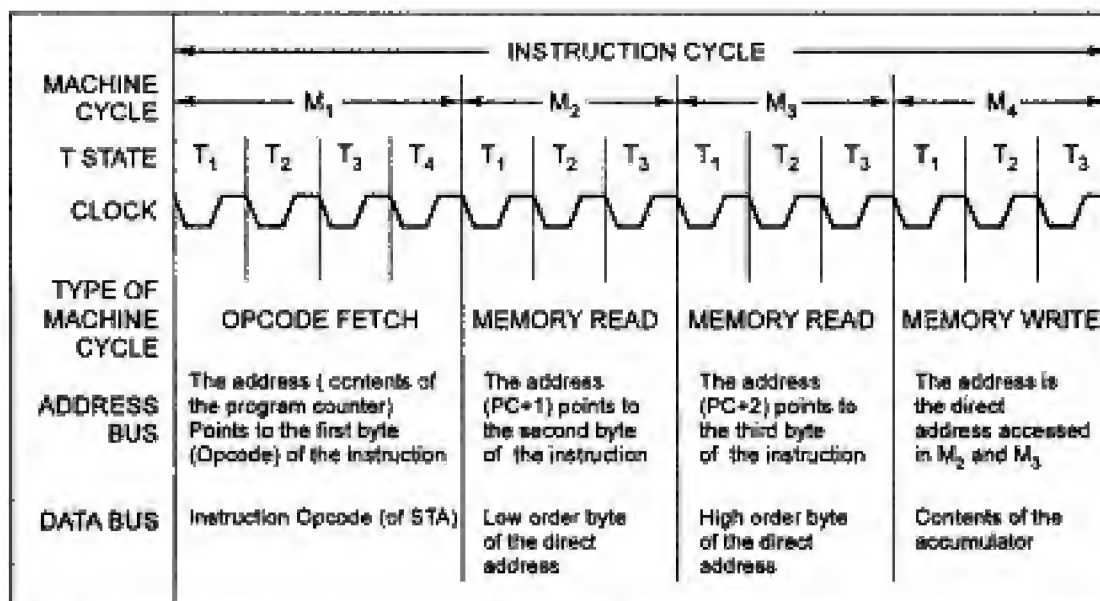


Fig. 2.42 Machine cycles for STA instruction

1. Opcode Fetch (4T- States).
2. Memory Read (3T- States).
3. Memory Read (3 T-States).
4. Memory Write (3 T-States).

Thus for STA instruction 13 T-States are required.

➡ **Example 2.3 :** Indicate machine cycles and T-States required for execution of LXI instruction.

Solution : For instruction shown in Fig. 2.43, three machine cycles are required to fetch the instruction. As it is an immediate instruction, operand i.e. immediate 16-bit data is given within the instruction, no further machine cycle is required.

Machine cycles required for LXI instruction.

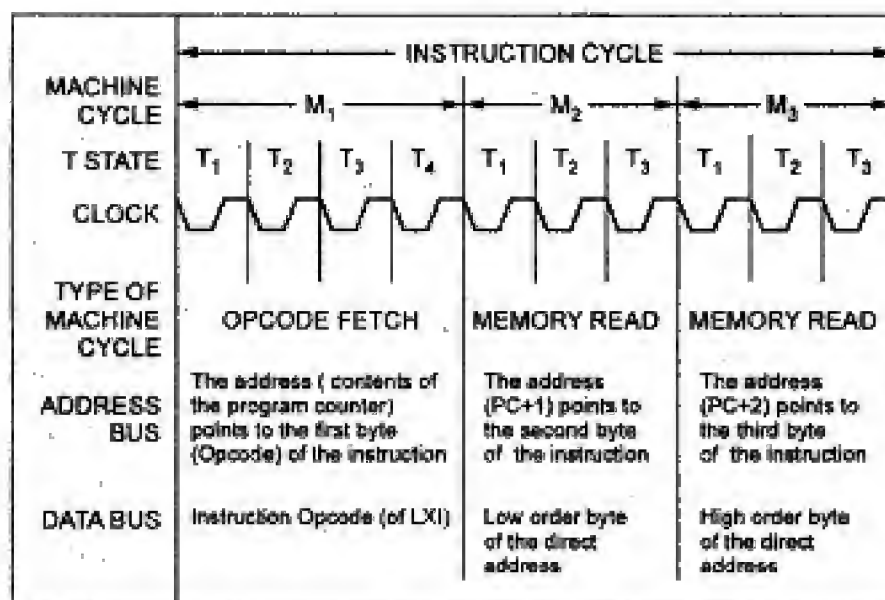


Fig. 2.43 Machine cycles required for LXI instruction(10)

1. Opcode Fetch (4T- States).
2. Memory Read (3T- States).
3. Memory Read (3T- States).

Thus for LXI instruction 10 T-States are required.

➡ **Example 2.4 :** Indicate machine cycles and T-States required for execution of LHLD instruction.

Solution : For instruction shown in Fig. 2.44, three machine cycles are required to fetch the instruction and two additional machine cycles are required to load 16-bit contents from two consecutive memory locations into HL register pair. So following machine cycles are required for LHLD instruction.

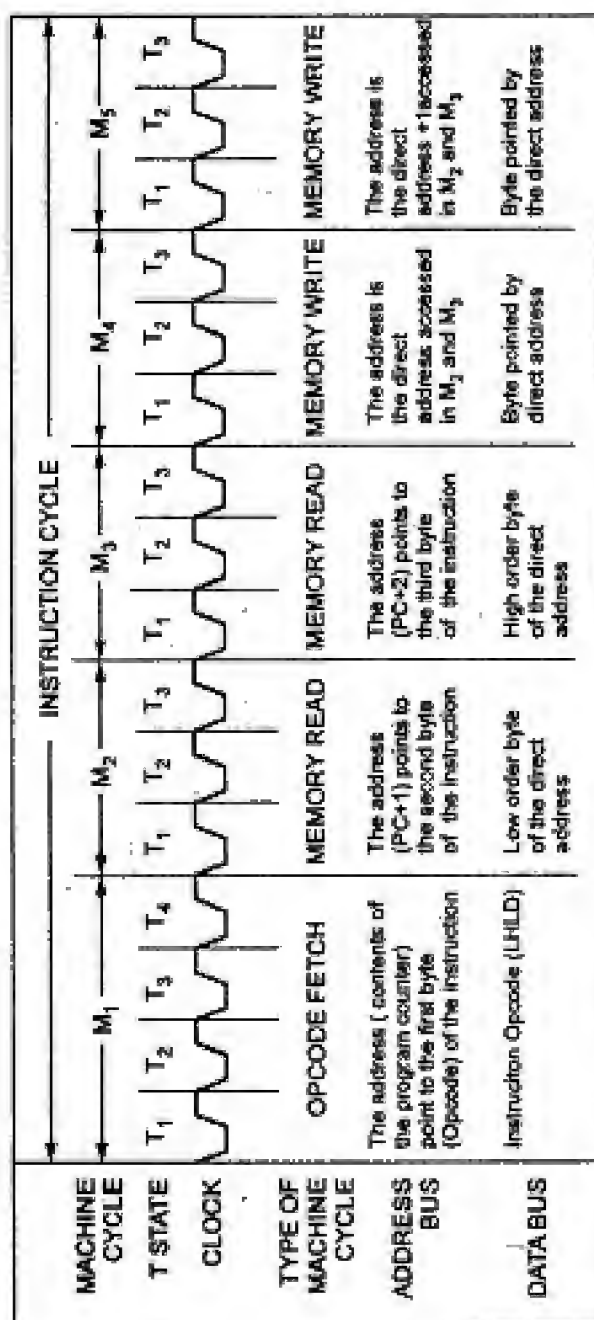


Fig. 2.44 Machine cycles and T-states required for execution of LHL D instruction

1. Opcode Fetch (4 T-States).
2. Memory Read (3 T-States).
3. Memory Read (3 T-States).
4. Memory Write (3 T-States).
5. Memory Write (3 T-States).

Thus for LHL D instruction 16T-States are required.

2.10.4 Concept of Wait States

In some applications, speed of memory system and I/O system are not compatible with the microprocessor's timings. This means that they take longer time to read/write data. In such situations, the microprocessor has to confirm whether a peripheral is ready to transfer data or not. If READY pin is high, the peripheral is ready otherwise 8085 enters wait state.

Fig. 2.45 shows the timing diagram for memory read machine cycle with and without wait state.

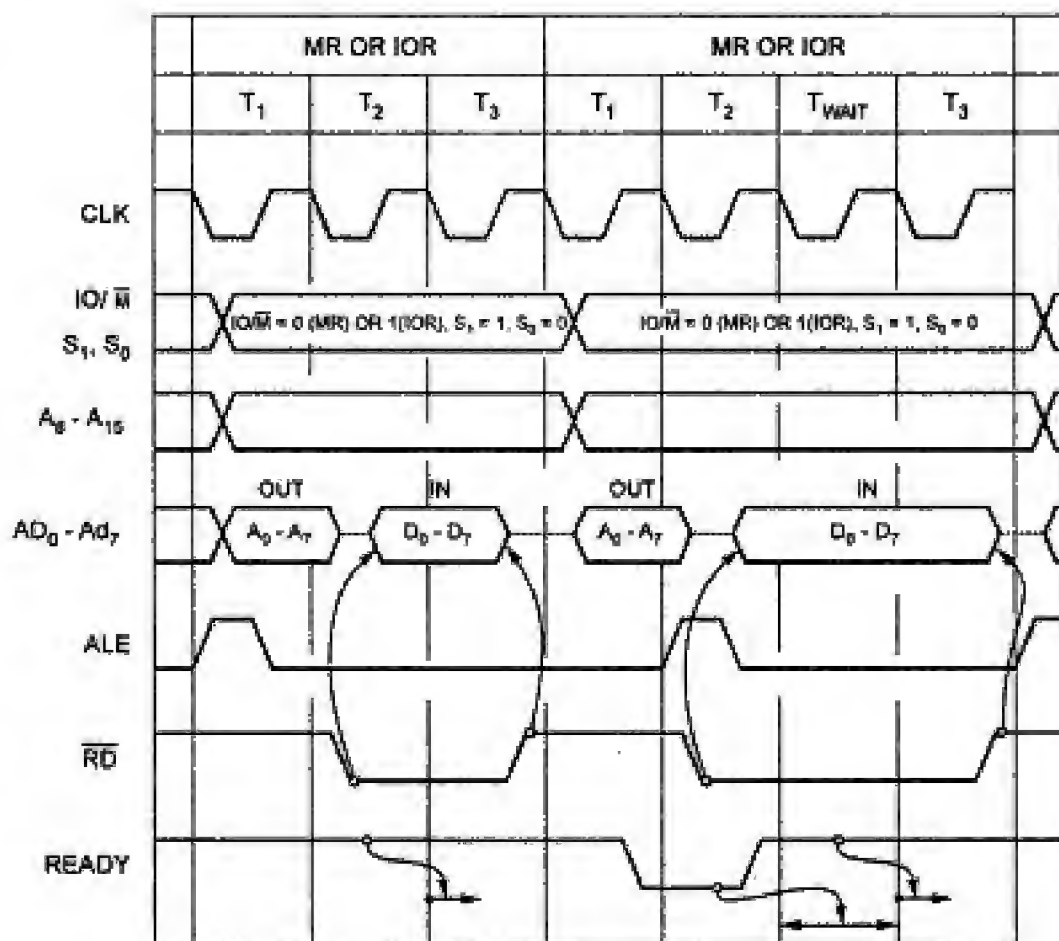


Fig. 2.45 Read machine cycle with and without wait state

Wait states continue to be inserted as long as READY is low. After the wait state, 8085 continues with T_3 of the machine cycle. During a wait state the contents of the address bus, the data bus, and the control bus are all held constant.

The wait state then gives an addressed memory or I/O port an extra clock cycle time to output valid data on the data bus. This feature allows to use cheaper memory or I/O devices that have longer access times.

Review Questions

1. Explain the features of 8085.
2. Give the clock out frequency and state time, T , of an 8085A operating with each of the following frequency crystals : 6.25 MHz, 6.144 MHz, 5 MHz and 4 MHz.
3. List the internal registers in 8085A, their abbreviations and lengths. Describe the primary function of each register.
4. Give the format of flag register in 8085. Explain each flag.
5. Draw the functional block diagram of microprocessor 8085 and explain in brief.
6. Explain different control signals used by 8085.
7. Why $AD_0 - AD_7$ lines are multiplexed ?
8. What is the use of ALE signal ?
9. What is the use of CLKOUT and RESET OUT signals of 8085 processor ?
10. Describe the function of following pins in 8085.
a. READY b. ALE c. $\overline{IO/M}$ d. HOLD e. RESET
11. Explain the signals used in DMA operation in 8085.
12. Draw and explain typical 8085 configuration.
13. Define
i) Instruction cycle ii) Machine cycle iii) T state
14. What is the necessity to have two status lines S_1 and S_0 in 8085 ?
15. Show the representation of single signal.
16. Show the representation of group of signals.
17. In which T - cycle the ALE signal is activated ?
18. Explain how the multiplexed data/address bus is shared for data and address.
19. Draw and explain the timing diagram of opcode fetch machine cycle.
20. Draw and explain the timing diagram of memory read machine cycle.
21. Draw and explain the timing diagram of memory write machine cycle.
22. Draw and explain the timing diagrams for I/O read and I/O write machine cycles.
23. Explain the interrupt acknowledge cycle for RST instruction.
24. Explain the interrupt acknowledge cycle for CALL instruction.
25. Draw and explain the bus idle machine cycle.
26. Indicate machine cycles and T-states required for execution of LDA instruction.
27. Explain the use of Ready signal in 8085.
28. Draw and explain the timing diagram for memory read machine cycle with and without wait state.



8085 Instruction Set and Assembly Language Programming

3.1 Introduction

In the previous chapter we have studied block diagram of microprocessor 8085. The block diagram shows microprocessor's functions for data processing and data handling. It also shows how each of these logic functions are connected together. Such microprocessor performs a particular task by executing proper sequence of instructions. Thus to perform a task in a particular microprocessor system, programmer has to know the instructions supported by microprocessor used in the microprocessor system.

This chapter explains the set of instructions supported by the 8085 microprocessor and explains how to write programs (set of instructions written in a proper sequence to perform a particular task) using them. This chapter also gives a large number of programs to perform different tasks.

3.2 Instruction Classification

The instructions provided by the 8085 can be categorised into five different groups based on the nature of function of the instructions.

- Data transfer operations
- Arithmetic operations
- Logical operations
- Branch operations and
- Stack, Input/Output and Machine control operations

3.2.1 Data Transfer Operations

The data transfer instructions load given data into register, copy data from register to register, copy data from register to memory location, and vice versa. In other words we can say that data transfer instructions copy data from source to destination. Source can be data or contents of register or contents of memory location whereas destination can be register or memory location. These instructions do not affect the flag register of the processor.

3.2.2 Arithmetic Operations

The arithmetic instructions provided by 8085 perform addition, subtraction, increment and decrement operations.

- **Addition** : Any 8-bit number, or the contents of a register, or the contents of a memory location can be added to the contents of the accumulator and the resulted sum is stored in the accumulator. The resulted carry bit is stored in the carry flag. In 8085, no two other registers can be added directly, i.e. the contents of B and C registers cannot be added directly. To add two 16-bit numbers the 8085 provides DAD instruction. It adds the data within the register pair to the contents of the HL register pair and resulted sum is stored in the HL register pair.
- **Subtraction** : Any 8-bit number, or the contents of a register, or the contents of a memory location can be subtracted from the contents of the accumulator and the result is stored in the accumulator. The resulted borrow bit is stored in the carry flag. In 8085, no two other registers can be added directly.
- **Increment/Decrement** : The 8085 has the increment and decrement instructions to increment and decrement the contents of any register, memory location or register pair by 1.

3.2.3 Logical Operations

The logical instructions provided by 8085 perform logical, rotate, compare and complement operations.

- **Logical** : Using logical instructions, any 8-bit number, or the contents of a register, or of a memory location can be logically ANDed, ORed, or Exclusive-ORed with the contents of the accumulator and the result is stored in the accumulator. The result also affects the flags according to definition of flags. For example, the zero result sets the zero flag.
- **Rotate** : These instructions allow shifting of each bit in the accumulator either left or right by 1 bit position.
- **Compare** : Any 8-bit number, or the contents of a register, or the contents of a memory location can be compared for equality, greater than, or less than, with the contents of the accumulator.
- **Complement** : The result of accumulator can be complemented with this instruction. It replaces all 0s by 1s and all 1s by 0s.

3.2.4 Branching Operations

These instructions allow the 8085 to change the sequence of the program, either unconditionally or under certain test conditions. These instructions include branch instructions, subroutine call and return instructions and restart instructions.

3.2.5 Stack, Input/Output and Machine Control Operations

These instructions control the stack operations, input/output operations and machine operations. The stack instructions allow the transfer of data from register pair to stack memory and from stack memory to the register pair. The input/output instruction allows the transfer of 8-bit data to input/output port. On the other hand machine instructions control the machine operations such as interrupt, halt, or do nothing.

3.3 Instruction and Data Format

Each instruction of the 8085 microprocessor has specific information fields. These information fields of instructions are called elements of instruction. These are :

- **Operation code :** The operation code field in the instruction specifies the operation to be performed. The operation is specified by binary code, hence the name operation code or simply opcode. For example, for 8085 processor operation code for ADD B instruction is 80H.
- **Source / destination operand :** The source/destination operand field directly specifies the source/destination operand for the instruction. In 8085, the instruction MOV A,B has B register contents as a source operand and A register contents as a destination operand because this instruction copies the contents of register B to register A.
- **Source operand address :** We know that the operation specified by the instruction may require one or more operands. The source operand may be in the 8085 register or in the memory. Many times the instruction specifies the address of the source operand so that operand(s) can be accessed and operated by the 8085 according to the instruction.

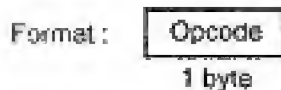
In 8085, the source operand address for instruction ADD M is given by HL register pair.

- **Destination operand address :** The operation executed by the 8085 may produce result. Most of the times the result is stored in one of the operand. Such operand is known as destination operand. The instruction which produce result specifies the destination operand address. In 8085, the destination operand address for instruction INR M is given by HL register pair because INR M instruction increments the contents of memory location specified by HL register pair and stores the result in the same memory location.
- **Next instruction address :** The next instruction address tells the 8085 from where to fetch the next instruction after completion of execution of current instruction. For BRANCH instructions the address of the next instruction is specified within the instruction. However, for other instructions, the next instruction to be fetched immediately follows the current instruction. For example, in 8085, instruction after INR B follows it. The instruction JMP 2000H specifies the next instruction address as 2000H.

3.3.1 Instruction Formats

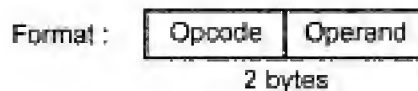
The 8085A instruction set consists of one, two and three byte instructions. The first byte is always the opcode; in two-byte instructions the second byte is usually data; in three byte instructions the last two bytes present address or 16-bit data.

1. One byte instruction :



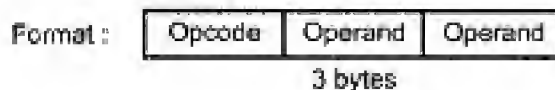
For Example : MOV A, B whose opcode is 78H which is one byte. This instruction copies the contents of B register in A register.

2. Two byte instruction :



For Example : MVI B, 02H. The opcode for this instruction is 06H and is always followed by a byte data (02H in this case). This instruction is a two byte instruction which copies immediate data into B register.

3. Three byte instruction :



For Example : JMP 6200H. The opcode for this instruction is C3H and is always followed by 16 bit address (6200H in this case). This instruction is a three byte instruction which loads 16 bit address into program counter.

3.3.2 Opcode Formats

The 8085A microprocessor has 8-bit opcodes. The opcode is unique for each instruction and contains the information about operation, register to be used, memory to be used etc. The 8085A identifies all operations, registers and flags with a specific code. For example, all internal registers are identified as shown in the tables 3.1(a) and 3.1(b).

| Registers | Code |
|------------|-------|
| B | 0 0 0 |
| C | 0 0 1 |
| D | 0 1 0 |
| E | 0 1 1 |
| H | 1 0 0 |
| L | 1 0 1 |
| M (Memory) | 1 1 0 |
| A | 1 1 1 |

Table 3.1(a)

| Register Pairs | Code |
|----------------|------|
| BC | 0 0 |
| DE | 0 1 |
| HL | 1 0 |
| AF or SP | 1 1 |

Table 3.1 (b)

Similarly, there are different codes for each operation. The operation codes are identified as follows :

| Instruction | Operation Code | | | | | | | | Instruction | Operation Code | | | | | | | |
|-------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|-------------|--|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
| | B ₇ | B ₆ | B ₅ | B ₄ | B ₃ | B ₂ | B ₁ | B ₀ | | B ₇ | B ₆ | B ₅ | B ₄ | B ₃ | B ₂ | B ₁ | B ₀ |
| ACI data | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | MVI r, data | 0 | 0 | D | D | D | 1 | 1 | 0 |
| ADC r | 1 | 0 | 0 | 0 | 1 | S | S | S | NOP | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| ADD r | 1 | 0 | 0 | 0 | 0 | S | S | S | ORA r | 1 | 0 | 1 | 1 | 0 | S | S | S |
| ADI data | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | ORI data | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 |
| ANA r | 1 | 0 | 1 | 0 | 0 | S | S | S | OUT addr | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 |
| ANI data | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | PCHL | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| CALL addr | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | POP rp | 1 | 1 | D | D | 0 | 0 | 0 | 1 |
| CMA | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | PUSH rp | 1 | 1 | D | D | 0 | 1 | 0 | 1 |
| CMC | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | R(Rotate) | 0 | 0 | 0 | A | D | 1 | 1 | 1 |
| | | | | | | | | | | D(Direction):L=0, R=1, A:Rotate All if A = 1 | | | | | | | |
| CMP r | 1 | 0 | 1 | 1 | 1 | S | S | S | RET | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 |
| DAA | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | RIM | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| DAD rp | 0 | 0 | D | D | 1 | 0 | 0 | 1 | RST n | 1 | 1 | n | n | n | 1 | 1 | 1 |
| DCR r | 0 | 0 | S | S | S | 1 | 0 | 1 | SBB r | 1 | 0 | 0 | 1 | 1 | S | S | S |
| DCX rp | 0 | 0 | D | D | 1 | 0 | 1 | 1 | SBI data | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 |
| DI | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | SHLD addr | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| EI | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | SIM | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| HLT | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | SPHL | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
| IN | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | STA addr | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| INR r | 0 | 0 | S | S | S | 1 | 0 | 0 | STAX rp | 0 | 0 | 0 | S | 0 | 0 | 1 | 0 |
| INX rp | 0 | 0 | D | D | 0 | 0 | 1 | 1 | STC | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 |
| JMP addr | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | SUB r | 1 | 0 | 0 | 1 | 0 | S | S | S |
| LDA addr | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | SUI data | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |
| LDAX rp | 0 | 0 | 0 | D | 1 | 0 | 1 | 0 | XCHG | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 |
| LHLD addr | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | XRA r | 1 | 0 | 1 | 0 | 1 | S | S | S |

| LXI rp | 0 | 0 | D | D | 0 | 0 | 0 | 1 | XRI data | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | | | | | | | | | | | | | | | |
|------------------|---|---|-----------------|---|---|---|---|---|--|---|---|---|---|---|---|---|-----------|---|---|-----------------|---|---|---|---|---|---|---|---|---|---|---|---|
| MOV rd, rs | D | 1 | D | D | D | S | S | S | | | | | | | | | | | | | | | | | | | | | | | | |
| J condition addr | 1 | 1 | T | T | F | 0 | 1 | 0 | <table><tr><th>Flag Type</th><th>T</th><th>T</th><th rowspan="5">F = Flag status</th></tr><tr><td>Z</td><td>0</td><td>0</td></tr><tr><td>C</td><td>0</td><td>1</td></tr><tr><td>P</td><td>1</td><td>0</td></tr><tr><td>S</td><td>1</td><td>1</td></tr></table> | | | | | | | | Flag Type | T | T | F = Flag status | Z | 0 | 0 | C | 0 | 1 | P | 1 | 0 | S | 1 | 1 |
| Flag Type | T | T | F = Flag status | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Z | 0 | 0 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| C | 0 | 1 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| P | 1 | 0 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| S | 1 | 1 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| C condition addr | 1 | 1 | T | T | F | 1 | 0 | 0 | | | | | | | | | | | | | | | | | | | | | | | | |
| R condition | 1 | 1 | T | T | F | 0 | 0 | 0 | | | | | | | | | | | | | | | | | | | | | | | | |

Note : DDD defines the destination register, SSS defines the source register, DD defines the register pair, D defines direction, TT defines Flag Type and F indicates flag status.

Table 3.2 Instruction codes

Now, we will see how opcodes are designed for some instructions using information given in the above tables.

Example 1 :

Instruction : MVI B Operation ; Move immediate 8-bit data
 Operation code : 0 0 D D D 1 1 0
 Register code : 0 0 0
 \therefore Opcode : 0 0 0 0 0 1 1 0 = 06H

Example 2 :

Instruction : MVI A Operation ; Move immediate 8-bit data
 Operation code : 0 0 D D D 1 1 0
 Register code : 1 1 1
 \therefore Opcode : 0 0 1 1 1 1 1 0 = 3EH

Example 3 :

Instruction : LXI B Operation ; Load immediate 16-bit data
 Operation code : 0 0 D D 0 0 0 1
 Register code : 0 0
 \therefore Opcode : 0 0 0 0 0 0 0 1 = 01H

Example 4 :

Instruction : LXI D Operation ; Load immediate 16-bit data
 Operation code : 0 0 D D 0 0 0 1
 Register code : 0 1
 \therefore Opcode : 0 0 0 1 0 0 0 1 = 11H

Example 5 :

Instruction : MOV A, B Operation ; Copy data from source register to destination register

Operation code : 0 1 D D D S S S

Register code : 1 1 1 0 0 0

∴ Opcode : 0 1 1 1 1 0 0 0 = 78H

Example 6 :

Instruction : MOV E, L Operation ; Copy data from source register to destination register

Operation code : 0 1 D D D S S S

Register code : 0 1 1 1 0 1

∴ Opcode : 0 1 0 1 1 1 0 1 = 5DH

3.3.3 Data Formats

The operand is another name for data. It may appear in different forms :

- Addresses
- Numbers/Logical data and
- Characters

Addresses : The address is a 16-bit unsigned integer number used to refer a memory location.

Numbers/Data : The 8085 supports following numeric data types.

- **Signed Integer :** A signed integer number is either a positive number or a negative number. In 8085, 8-bits are assigned for signed integer, in which most significant bit is used for sign and remaining seven bits are used for magnitude. Sign bit-0 indicates positive number whereas sign bit-1 indicates negative number.
- **Unsigned Integer :** The 8085 microprocessor supports 8-bit unsigned integer.
- **BCD :** The term BCD number stands for binary coded decimal number. It uses ten digits from 0 through 9. The 8-bit register of 8085 can store two digit BCD number.

Characters : The 8085 uses ASCII code to represent characters. It is a 7-bit alphanumeric code that represents decimal numbers, English alphabets, and other special characters.

3.4 Instruction Set of 8085

In this section, the instructions from all groups are explained with the help of examples. Before to discuss these instructions, let us get familiar with the notations used in the explanation of instructions. These are:

| Notation | Meaning |
|----------|---|
| M | Memory location pointed by HL register pair |
| r | 8-bit register |
| rp | 16-bit register pair |
| rs | Source register |
| rd | Destination register |
| addr | 16-bit address/8-bit address |

3.4.1 Data Transfer Group

1. **MVI r, data (8)** This instruction directly loads a specified register with an 8-bit data given within the instruction. The register r is an 8-bit general purpose register such as A, B, C, D, E, H and L.

Operation : $r \leftarrow \text{8-bit data (byte)}$

Example :

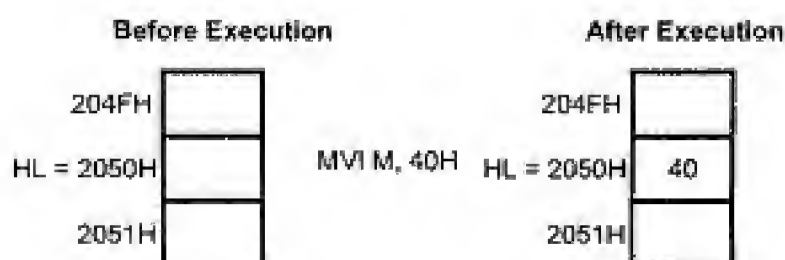
MVI B, 60H ; This instruction will load 60H directly into the B register.

2. **MVI M, data (8)** This instruction directly loads an 8-bit data given within the instruction into a memory location. The memory location is specified by the contents of HL register pair.

Operation : $M \leftarrow \text{byte or } (HL) \leftarrow \text{byte}$

Example : H = 20H and L = 50H

MVI M, 40H ; This instruction will load 40H into memory whose address is 2050H.



3. **MOV rd, rs** This instruction copies data from the source register into destination register. The rs and rd are general purpose registers such as A, B, C, D, E, H and L. The contents of the source register remain unchanged after execution of the instruction.

Operation : $rd \leftarrow rs$

Example : $A = 20H$

MOV B, A ; This instruction will copy the contents of register A (20H) into register B.

4. **MOV M, rs** This instruction copies data from the source register into memory location pointed by the HL register pair. The rs is an 8-bit general purpose register such as A, B, C, D, E, H and L.

Operation : $(HL) \leftarrow rs$

Example : If $HL = 2050H$, $B = 30H$.

MOV M, B ; This instruction will copy the contents of B register (30H) into the memory location whose address is specified by HL (2050H).

5. **MOV rd, M** This instruction copies data from memory location whose address is specified by HL register pair into destination register. The contents of the memory location remain unchanged. The rd is an 8-bit general purpose register such as A, B, C, D, E, H and L.

Operation : $rd \leftarrow (HL)$

Example : $HL = 2050H$, contents at 2050H memory location = 40H

MOV C, M ; This instruction will copy the contents of memory location pointed by HL register pair (40H) into the C register.

6. **LXI rp, data (16)** This instruction loads immediate 16-bit data specified within the instruction into register pair or stack pointer. The rp is 16-bit register pair such as BC, DE, HL or 16-bit stack pointer.

Operation : $rp \leftarrow \text{data (16)}$

Example :

LXI B, 1020H ; This instruction will load 10H into B register and 20H into C register.

- 7. STA addr** This instruction stores the contents of A register into the memory location whose address is directly specified within the instruction. The contents of A register remain unchanged.
- Operation** : $(addr) \leftarrow A$
- Example** : $A = 50H$
- STA 2000H ; This instruction will store the contents of A register (50H) to memory location 2000H.
- 8. LDA addr** This instruction copies the contents of the memory location whose address is given within the instruction into the accumulator. The contents of the memory location remain unchanged.
- Operation** : $A \leftarrow (addr)$
- Example** : $(2000H) = 30H$
- LDA 2000H ; This instruction will copy the contents of memory location 2000H i.e. data 30H into the A register
- 9. SHLD addr** This instruction stores the contents of L register in the memory location given within the instruction and contents of H register at address next to it. This instruction is used to store the contents of H and L registers directly into the memory. The contents of the H and L registers remain unchanged.
- Operation** : $(addr) \leftarrow L$ and $(addr + 1) \leftarrow H$
- Example** : $H = 30H, L = 60H$
- SHLD 2500H ; This instruction will copy the contents of L register at address 2500H and the contents of H register at address 2501H.
- 10. LHLD addr** This instruction copies the contents of the memory location given within the instruction into the L register and the contents of the next memory location into the H register.
- Operation** : $L \leftarrow (addr), H \leftarrow (addr + 1)$
- Example** : $(2500H) = 30H, (2501H) = 60H$
- LHLD 2500H ; This instruction will copy the contents of memory location 2500H i.e. data 30H into the L register and the contents at memory location 2501H i.e. data 60H into the H register.

11. STAX rp This instruction copies the contents of accumulator into the memory location whose address is specified by the specified register pair. The rp is BC or DE register pair. This register pair is used as a memory pointer. The contents of the accumulator remain unchanged.

Operation : $(rp) \leftarrow A$

Example : BC = 1020H, A = 50H

STAX B ; This instruction will copy the contents of A register (50H) to the memory location specified by BC register pair (1020H).

12. LDAX rp This instruction copies the contents of memory location whose address is specified by the register pair into the accumulator. The rp is BC or DE register pair. The register pair is used as a memory pointer.

Operation : $A \leftarrow (rp)$

Example : DE = 2030H, (2030H) = 80H

LDAX D This instruction will copy the contents of memory location specified by DE register pair (80H) into the accumulator.

13. XCHG This instruction exchanges the contents of the register H with that of D and of L with that of E.

Operation : $H \leftrightarrow D$ and $L \leftrightarrow E$

Example : DE = 2040H, HL = 7080H

XCHG ; This instruction will load the data into registers as follows H = 20H, L = 40H, D = 70H and E = 80

3.4.2 Arithmetic Group

1. ADD r This instruction adds the contents of the specified register to the contents of accumulator and stores result in the accumulator. The r is 8-bit general purpose register such as A, B, C, D, E, H and L.

Operation : $A \leftarrow A + r$

Example : A = 20H, C = 30H.

ADD C ; This instruction will add the contents of C register, i.e. data 30H to the contents of accumulator, i.e. data 20H and it will store the result 50H in the accumulator.

2. ADD M This instruction adds the contents of the memory location pointed by HL register pair to the contents of accumulator and stores result in the accumulator. The HL register pair is used as a memory pointer. This instruction affects all flags.

Operation : $A \leftarrow A + M$

Example : A = 20H, HL = 2050H, (2050H) = 10H

ADD M ; This instruction will add the contents of memory location pointed by HL register pair, 2050H i.e. data 10H to the contents of accumulator i.e. data 20H and it will store the result, 30H in the accumulator.

3. ADI data (8) This instruction adds the 8 bit data given within the instruction to the contents of accumulator and stores the result in the accumulator.

Operation : $A \leftarrow A + \text{data (8)}$

Example : A = 50H

ADI 70H ; This instruction will add 70H to the contents of the accumulator (50H) and it will store the result in the accumulator (C0H).

4. ADC r This instruction adds the contents of specified register to the contents of accumulator with carry. This means, if the carry flag is set by some previous operation, it adds 1 and the contents of the specified register to the contents of accumulator, else it adds the contents of the specified register only. The r is 8-bit general purpose register such as A, B, C, D, E, H and L.

Operation : $A \leftarrow A + r + CY$

Example : Carry flag = 1, A = 50H, C = 20H

ADC C ; This instruction will add the contents of C (20H) register to the contents of accumulator (50H) with carry (1) and it will store result, 71H (50H + 20H + 1 = 71H) in the accumulator

- 5. ADC M** This instruction adds the contents of memory location pointed by HL register pair to the contents of accumulator with carry and stores the result in the accumulator. HL register pair is used as a memory pointer.

Operation : $A \leftarrow A + M + CY$

Example : Carry flag = 1, HL = 2050H, A = 20H, (2050H) = 30H.

ADC M ; This instruction will add the contents of memory location pointed by HL register pair, 2050H, i.e. data 30H to the contents of accumulator, i.e. data 20H with carry flag (1). It will store the result (30+20+1=51H) in the accumulator.

- 6. ACI data (8)** This instruction adds 8 bit data given within the instruction to the contents of accumulator with carry and stores result in the accumulator.

Operation : $A \leftarrow A + \text{data (8)} + CY$

Example : A = 30H, Carry flag = 1

ACI 20H ; This instruction will add 20H to the contents of accumulator, i.e. data 30H with carry (1) and stores the result, 51H (30 + 20 + 1 = 51H) in the accumulator.

- 7. DAD rp** This instruction adds the contents of the specified register pair to the contents of the HL register pair and stores the result in the HL register pair. The rp is 16-bit register pair such as BC, DE, HL or stack pointer. Only higher order register is to be specified for register pair within the instruction.

Operation : $HL \leftarrow HL + rp$

Example : DE = 1020H, HL = 2050H

DAD D ; This instruction will add the contents of DE register pair, 1020H to the contents of HL register pair, 2050H. It will store the result, 3070H in the HL register pair.

- 8. SUB r** This instruction subtracts the contents of the specified register from the contents of the accumulator and stores the result in the accumulator. The register r is 8-bit general purpose register such as A, B, C, D, E, H and L.

Operation : $A \leftarrow A - r$

Example : $A = 50H, B = 30H.$

SUB B ; This instruction will subtract the contents of B register (30H) from the contents of accumulator (50H) and stores the result (20H) in the accumulator.

9. SUB M This instruction subtracts the contents of the memory location pointed by HL register pair from the contents of accumulator and stores the result in the accumulator. The HL register pair is used as a memory pointer.

Operation : $A \leftarrow A - M$

Example : $HL = 1020H, A = 50H, (1020H) = 10H$

SUB M ; This instruction will subtract the contents of memory location pointed by HL register pair, 1020H, i.e. data 10H from the contents accumulator, i.e. data 50H and stores the result (40H) in accumulator.

10. SUI data (8) This instruction subtracts an 8-bit data given within the instruction from the contents of the accumulator and stores the result in the accumulator.

Operation : $A \leftarrow A - \text{data (8)}$

Example : $A = 40H,$

SUI 20H ; This instruction will subtract 20H from the contents of accumulator (40H). It will store the result (20H) in the accumulator.

11. SBB r This instruction subtracts the specified register contents and borrow flag from the accumulator contents. This means, if the carry flag (borrow for subtraction) is set by some previous operation, it subtracts 1 and the contents of the specified register from the contents of accumulator, else it subtracts the contents of the specified register only. The register r is 8-bit register such as A, B, C, D, E, H and L.

Operation : $A \leftarrow A - r - CY$

Example : Carry flag = 1, C = 20H, A = 40H

SBB C ; This instruction will subtract the contents of C register (20H) and carry flag (1) from the contents of accumulator (40H). It will store the result $(40H - 20H - 1 = 1FH)$ in the accumulator.

12. SBB M This instruction subtracts the contents of memory location pointed by HL register pair from the contents of accumulator and borrow flag and stores the result in the accumulator.

Operation : $A \leftarrow A - M - CY$

Example : Carry flag = 1, HL = 2050H, A = 50H, (2050H) = 10H.

SBB M ; This instruction will subtract the contents of memory location: pointed by HL register pair, 2050H, i.e. data 10H and borrow (Carry flag = 1) from the contents of accumulator (50H) and stores the result 3FH in the accumulator $(50 - 10 - 1 = 3F)$.

13. SBI data (8) This instruction subtracts 8 bit data given within the instruction and borrow flag from the contents of accumulator and stores the result in the accumulator.

Operation : $A \leftarrow A - \text{data}(8) - CY$

Example : Carry flag = 1, A = 50H

SBI 20H ; This instruction will subtract 20H and the carry flag (1) from the contents of the accumulator (50H). It will store the result $(50H - 20H - 1 = 2FH)$ in the accumulator.

14. DAA This instruction adjusts accumulator to packed BCD (Binary Coded Decimal) after adding two BCD numbers.

Instruction works as follows :

1. If the value of the low-order four bits ($D_3 - D_0$) in the accumulator is greater than 9 or if auxiliary carry flag is set, the instruction adds 6 (06) to the low-order four bits.
2. If the value of the high-order four bits ($D_7 - D_4$) in the accumulator is greater than 9 or if carry flag is set, the instruction adds 6(60) to the high-order four bits.

Example :

```

If,      A = 0011 1001 = 39 BCD
and      C = 0001 0010 = 12 BCD then
      ADD C      ; Gives A = 0100 1011 = 4BH
      DAA        ; adds 0110 because 1011 > 9,
                  ; A=0101 0001 = 51 BCD
If      A = 1001 0110 = 96 BCD
and      D = 0000 0111 = 07 BCD then
      ADD D      ; Gives A = 1001 1101 = 9DH
      DAA        ; adds 0110 because 1101 > 9,
                  ; A = 1010 0011 = A3H,
                  ; 1010 > 9 so adds 0110 0000,
                  ; A = 0000 0011 = 03 BCD, CF = 1.

```

15. INR r This instruction increments the contents of specified register by 1. The result is stored in the same register. The register r is 8-bit general purpose register such as A, B, C, D, E, H and L.

Operation : $r \leftarrow r + 1$

Example : B = 10H

INR B ; This instruction will increment the contents of B register (10H) by one and stores the result (10 + 1 = 11H) in the same i.e. B register.

16. INR M This instruction increments the contents of memory location pointed by HL register pair by 1. The result is stored at the same memory location. The HL register pair is used as a memory pointer.

Operation : $M \leftarrow M + 1$

Example : HL = 2050H, (2050H) = 30H

INR M ; This instruction will increment the contents of memory location pointed by HL register pair, 2050H, i.e. data 30H by one. It will store the result (30 + 1 = 31H) at the same place.

17. INX rp This instruction increments the contents of register pair by one. The result is stored in the same register pair. The rp is register pair such as BC, DE, HL or stack pointer (SP).

Operation : $rp \leftarrow rp + 1$

- Example** : HL = 10FFH
- INX H** ; This instruction will increment the contents of HL register pair (10FFH) by one. It will store the result (10FF + 1 = 1100H) in the same i.e. HL register pair.
- 18. DCR r** This instruction decrements the contents of the specified register by one. It stores the result in the same register. The register r is 8-bit general purpose register such as A, B, C, D, E, H and L.
- Operation** : $r \leftarrow r - 1$
- Example** : E = 20H
- DCR E** ; This instruction will decrement the contents of E register (20H) by one. It will store the result (20 - 1 = 1FH) in the same, i.e. E register.
- 19. DCR M** This instruction decrements the contents of memory location pointed by HL register pair by 1. The HL register pair is used as a memory pointer. The result is stored in the same memory location.
- Operation** : $M \leftarrow M - 1$
- Example** : HL = 2050H, (2050H) = 21H
- DCR M** ; This instruction will decrement the contents of memory location pointed by HL register pair, 2050H, i.e. data 21H by one. It will store the result (21 - 1 = 20H) in the same memory location.
- 20. DCX rp** This instruction decrements the contents of register pair by one. The result is stored in the same register pair. The rp is register pair such as BC, DE, HL or stack pointer (SP). Only higher order register is to be specified within the instruction.
- Operation** : $rp \leftarrow rp - 1$
- Example** : DE = 1020H
- DCX D** ; This instruction will decrement the contents of DE register pair (1020H) by one and store the result (1020 - 1 = 101FH) in the same, DE register pair.

3.4.3 Logic Group

1. **ANA r** This instruction logically ANDs the contents of the specified register with the contents of accumulator and stores the result in the accumulator. Each bit in the accumulator is logically ANDed with the corresponding bit in register r, i.e. D_0 bit in A with D_0 bit in register r, D_1 in A with D_1 in r and so on upto D_7 bit. The register r is 8-bit general purpose register such as A, B, C, D, E, H and L.

Operation : $A \leftarrow A \wedge r$

Example : A = 10101010 (AAH), B = 00001111 (0FH)

ANA B ; This instruction will logically AND the contents of B register with the contents of accumulator. It will store the result (0AH) in the accumulator.

1010 1010
0000 1111

0000 1010 = 0AH

2. **ANA M** This instruction logically ANDs the contents of memory location pointed by HL register pair with the contents of accumulator. The result is stored in the accumulator. The HL register pair is used as a memory pointer.

Operation : $A \leftarrow A \wedge M$

Example : A = 01010101 = (55H), HL = 2050H(2050H) \rightarrow 10110011 = (B3H)

ANA M ; This instruction will logically AND the contents of memory location pointed by HL register pair (B3H) with the contents of accumulator (55H). It will store the result (11H) in the accumulator

0101 0101
1011 0011

0001 0001 = 11H

3. **ANI data** This instruction logically ANDs the 8 bit data given in the instruction with the contents of the accumulator and stores the result in the accumulator.

Operation : $A \leftarrow A \wedge \text{data (B)}$

Example : A = 1011 0011 = (B3H)

ANI 3FH ; This instruction will logically AND the contents of
1011 0011 accumulator (B3H) with 3FH. It will store the result
0011 1111 (33H) in the accumulator.

0011 0011 = 33H

The AND operation clears bits of a binary number. The task of clearing a bit in a binary number is called **masking**. The Fig. 3.1 shows the process of masking.

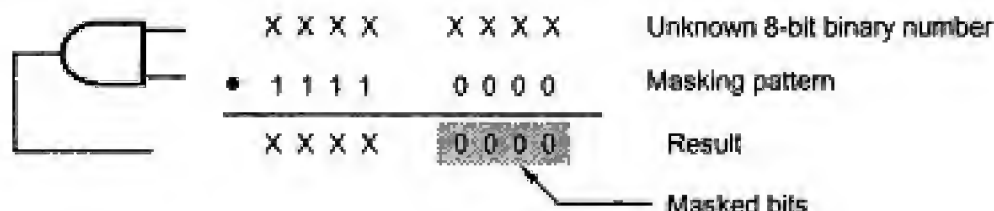


Fig. 3.1 Masking using AND operation

4. **XRA r** This instruction logically XORs the contents of the specified register with the contents of accumulator and stores the result in the accumulator. The register *r* is 8-bit general purpose register such as A, B, C, D, E, H and L.

Operation : $A \leftarrow A \oplus r$

Example : A = 1010 1010 (AAH), C = 0010 1101 (2DH)

XRA C ; This instruction will logically XOR the contents of C
1010 1010 register with the contents of accumulator. It will
0010 1101 store the result (87H) in the accumulator.

1000 0111 = (87H)

5. **XRA M** This instruction logically XORs the contents of memory location pointed by HL register pair with the contents of accumulator. The HL register pair is used as a memory pointer.

Operation : $A \leftarrow A \oplus M$

Example : A = 0101 0101 = (55H), HL = 2050H (2050H) → 1011 0011
= (B3H)

XRA M ; This instruction will logically XOR the contents of
0101 0101 memory location pointed by HL register pair (2050H)
1011 0011 i.e. data B3H with the contents of accumulator (55H).
It will store the result (E6H) in the accumulator.

1110 0110 = E6H

6. **XRI data** This instruction logically XORs the 8 bit data given in the instruction with the contents of the accumulator and stores the result in the accumulator.

Operation : $A \leftarrow A \oplus \text{data}$

Example : $A = 10110011 = (B3H)$

XRI 39H ; This instruction will logically XOR the contents of
1011 0011 accumulator (B3H) with 39H. It will store the result
0011 1001 (8AH) in the accumulator.

1000 1010 = 8AH

The XOR instruction is used if some bits of a register or memory location must be inverted. This instruction allows part of a number to be inverted or complemented. This is illustrated in Fig. 3.2.

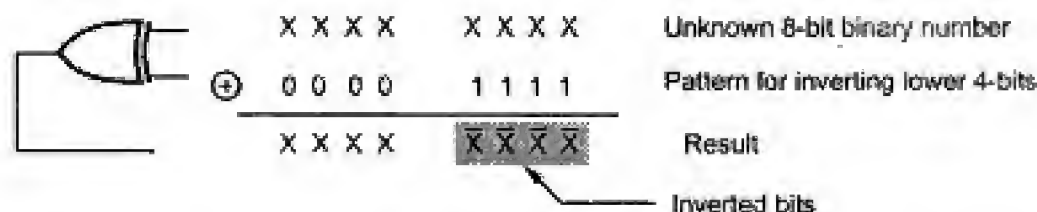


Fig. 3.2 Inversion of part of a number using XOR operation

7. **ORA r** This instruction logically ORs the contents of specified register with the contents of accumulator and stores the result in the accumulator. Each bit in the accumulator is ORed with corresponding bit in register r. i.e. D_0 bit in accumulator is ORed with D_0 bit in register r, D_1 in A with D_1 in r and so on upto D_7 bit. The register r is 8-bit general purpose register such as A, B, C, D, E, H and L.

Operation : $A \leftarrow A \vee r$

Example : $A = 1010 1010$ (AAH), $B = 0001 0010$ (12H)

ORA B ; This instruction will logically OR the contents of B
1010 1010 register with the contents of accumulator. It will
0001 0010 store the result (BAH) in the accumulator.

1011 1010 = BAH

- 8. ORA M** This instruction logically ORs the contents of memory location pointed by HL register pair with the contents of accumulator. The result is stored in the accumulator. The HL register pair is used as a memory pointer.

Operation : $A \leftarrow A \vee M$

Example : $A = 0101\ 0101 = (55H)$ $HL = 2050H$

$(2050H) \rightarrow 1011\ 0011 = (B3H)$

ORA M ; This instruction will logically OR the contents of
 0101 0101 memory location pointed by HL register pair (B3H) with
 1011 0011 the contents of accumulator (55H). It will store the
 result (F7H) in the accumulator.

1111 0111 = F7H

- 9. ORI data** This instruction logically ORs the 8 bit data given in the instruction with the contents of the accumulator and stores the result in the accumulator.

Operation : $A \vee \text{data (8)}$

Example : $A = 1011\ 0011 = (B3H)$

ORI 08H ; This instruction will logically OR the contents of
 1011 0011 accumulator (B3H) with 08H. It will store the result
 0000 1000 (BBH) in the accumulator.

1011 1011 = BBH

The OR instruction is used to set (make one) any bit in the binary number. This is illustrated in Fig. 3.3.

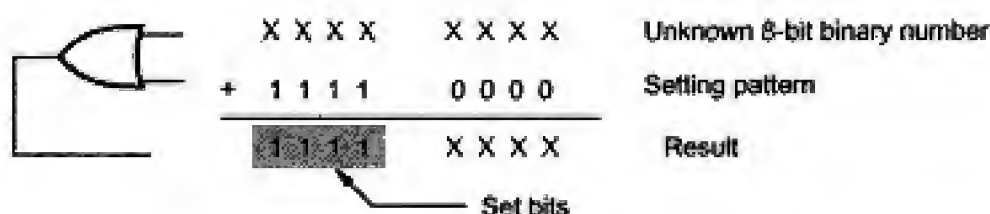


Fig. 3.3 Setting bit/s using OR operation

- 10. CMP r** This instruction subtracts the contents of the specified register from contents of the accumulator and sets the condition flags as a result of the subtraction. It sets zero flag if $A = r$ and sets carry flag if $A < r$. The register r is 8-bit general purpose register such as A, B, C, D, E, H and L.

Operation : $A - r$

Example : A = 1011 1000 (B8H) and D = 1011 1001 (B9H)

CMP D ; This instruction will compare the contents of D register with the contents of accumulator. Here A < D so carry flag will set after the execution of the instruction.

11. CMP M This instruction subtracts the contents of the memory location specified by HL register pair from the contents of the accumulator and sets the condition flags as a result of subtraction. It sets zero flag if A = M and sets carry flag if A < M. The HL register pair is used as a memory pointer.

Operation : A - M

Example : A = 1011 1000 (B8H), HL = 2050H and (2050H) = 1011 1000 (B8H)

CMP M ; This instruction will compare the contents of memory location (B8H) and the contents of accumulator. Here A = M so zero flag will set after the execution of the instruction.

12. CPI data This instruction subtracts the 8 bit data given in the instruction from the contents of the accumulator and sets the condition flags as a result of subtraction. It sets zero flag if A = data and sets carry flag if A < data.

Operation : A - data (8)

Example : A = 1011 1010 = (BAH)

CPI 30H ; This instruction will compare 30H with the contents of accumulator (BAH). Here A > data so zero and carry both flags will reset after the execution of the instruction.

13. STC This instruction sets carry flag = 1

Operation : CY ← 1

Example : Carry flag = 0

STC ; This instruction will set the carry flag = 1

14. CMC This instruction complements the carry flag.

Operation : $CY \leftarrow \overline{CY}$

Example : Carry flag = 1

CMC ; This instruction will complement the carry flag i.e. carry flag = 0

15. CMA This instruction complements each bit of the accumulator.

Operation : $A \leftarrow \overline{A}$

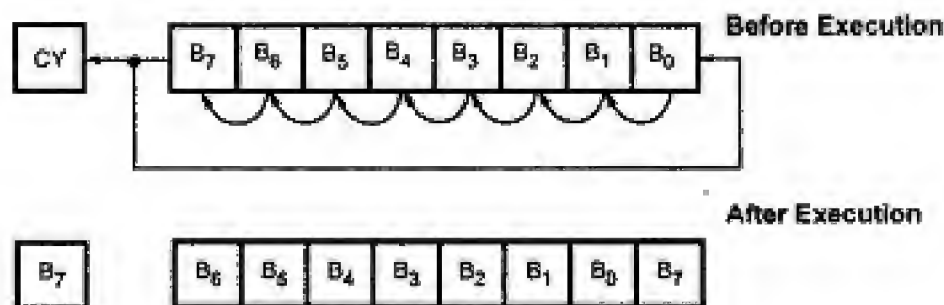
Example : $A = 1000\ 1000 = 88H$

CMA ; This instruction will complement each bit of accumulator $A = 0111\ 0111 = 77H$

3.4.4 Rotate Group

1. RLC This instruction rotates the contents of the accumulator left by one position. Bit B_7 is placed in B_0 as well as in CY.

Operation :

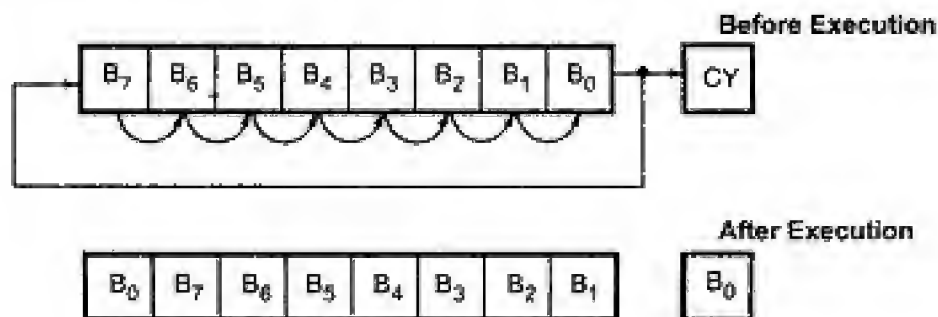


Example : $A = 01010111$ (57H) and $CY = 1$

RLC ; After execution of the instruction the accumulator contents will be (1010 1110) AEH and carry flag will reset.

2. RRC This instruction rotates the contents of the accumulator right by one position. Bit B_0 is placed in B_7 as well as in CY.

Operation :

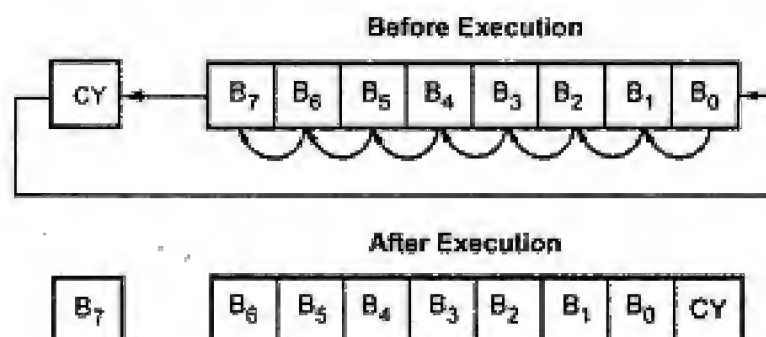


Example : A = 1001 1010 (9AH) and CY = 1

RRC ; After execution of the instruction the accumulator contents will be (0100 1101) 4DH and carry flag will reset.

3. RAL This instruction rotates the contents of the accumulator left by one position. Bit B₇ is placed in CY and CY is placed in B₀.

Operation :

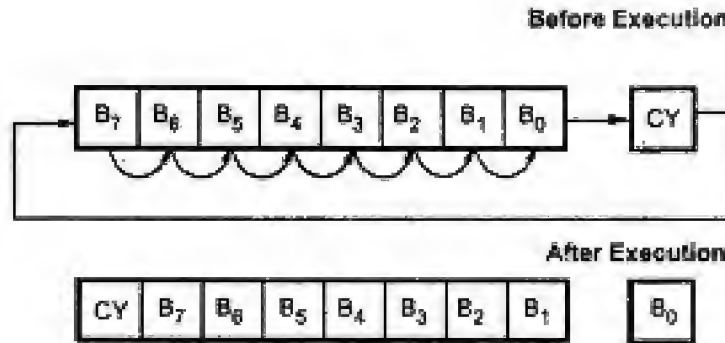


Example : A = 10101101 (ADH) and CY = 0

RAL ; After execution of the instruction accumulator contents will be (0101 1010) 5AH and carry flag will set.

4. RAR This instruction rotates the contents of the accumulator right by one position. Bit B₀ is placed in CY and CY is placed in B₇.

Operation :



Example : A = 1010 0011 (A3H) and CY = 0

RAR ; After execution of the instruction accumulator contents will be (0101 0001) 51H and carry flag will set.

3.4.5 Stack Operations

The stack is a portion of read/write memory set aside by the user for the purpose of storing information temporarily. When the information is written on the stack, the operation is called PUSH. When the information is read from stack, the operation is called POP.

The microprocessor stores the information, much like stacking plates. Using this analogy of stacking plates it is easy to illustrate the stack operation.



Fig. 3.4 Stacked plates

Fig. 3.4 shows the stacked plates. Here, we realize that if it is desired to take out the first stacked plate we will have to remove all plates above the first plate in the reverse order. This means that to remove first plate we will have to remove the third plate, then the second plate and finally the first plate. This means that, the first information pushed on to the stack is the last information popped off from the stack. This type of operation is known as a first in, last out (FILO). This stack is

implemented with the help of special memory pointer register. The special pointer register is called the **stack pointer**. During PUSH and POP operation, stack pointer register gives the address of memory where the information is to be stored or to be read. The stack pointer's contents are automatically manipulated to point to stack top. The memory location currently pointed by stack pointer is called **top of stack**.

1. PUSH rp

This instruction decrements stack pointer by one and copies the higher byte of the register pair into the memory location pointed by stack pointer. It then decrements the stack pointer again by one and copies the lower byte of the register pair into the memory location pointed by stack pointer. The rp is 16-bit register pair such as BC, DE, HL. Only higher order register is to be specified within the instruction.

Operation : $SP \leftarrow SP - 1$, $(SP) \leftarrow rpH$, $SP \leftarrow SP - 1$, $(SP) \leftarrow rpL$.

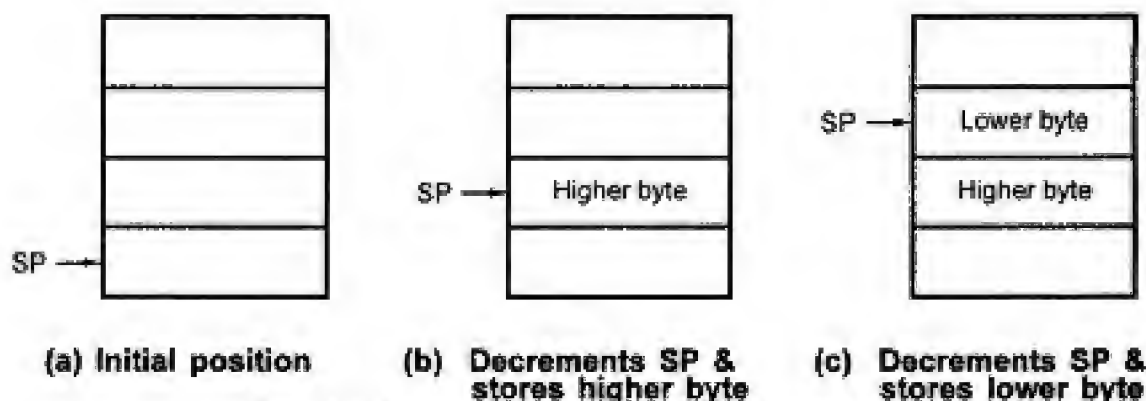
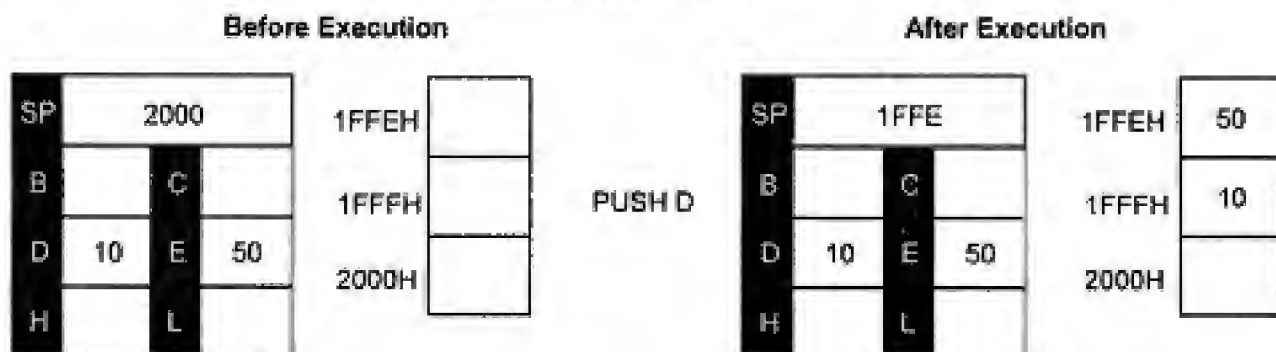


Fig. 3.5 Steps Involved in PUSH Operation

Example : $SP = 2000H$, $DE = 1050H$.

PUSH D : This instruction will decrement the stack pointer ($2000H$) by one ($SP = 1FFFH$) and copies the contents of D register ($10H$) into the memory location $1FFFH$. It then decrements the stack pointer again by one ($SP = 1FFE$) and copies the contents of E register ($50H$) into the memory location $1FFE$.



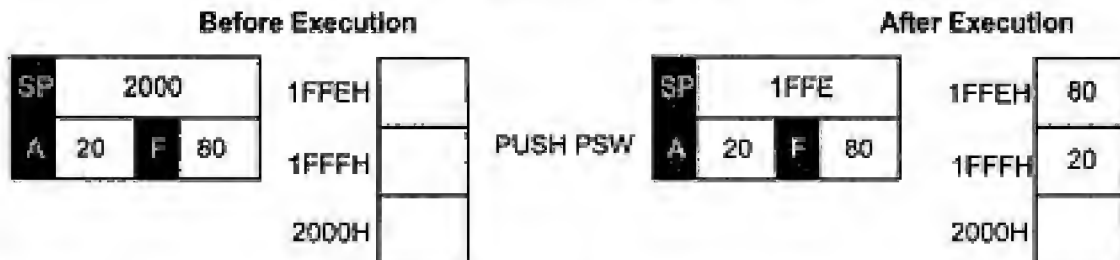
2. PUSH PSW

This instruction decrements stack pointer by one and copies the accumulator contents into the memory location pointed by stack pointer. It then decrements the stack pointer again by one and copies the flag register into the memory location pointed by the stack pointer.

Operation : $SP \leftarrow SP - 1$
 $(SP) \leftarrow A$
 $SP \leftarrow SP - 1$
 $(SP) \leftarrow \text{Flag register}$

Example : SP = 2000H, A = 20H, Flag register = 80H

PUSH PSW ; This instruction decrements the stack pointer (SP = 2000H) by one (SP = 1FFFH) and copies the contents of the accumulator (20H) into the memory location 1FFFH. It then decrements the stack pointer again by one (SP = 1FFE) and copies the contents of the flag register (80H) into the memory location 1FFE.



3. POP rp

This instruction copies the contents of memory location pointed by the stack pointer into the lower byte of the specified register pair and increments the stack pointer by one. It then copies the contents of memory location pointed by stack pointer into the higher byte of the specified register pair and increments the stack pointer again by one. The rp is 16-bit register pair such as BC, DE, HL. Only higher order register is to be specified within the instruction.

Operation : $rpL \leftarrow (SP)$
 $SP \leftarrow SP + 1$
 $rpH \leftarrow (SP), SP \leftarrow SP + 1$

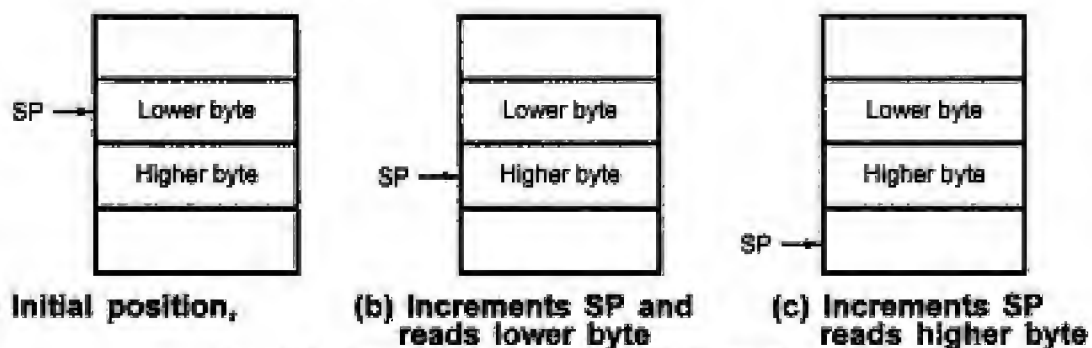
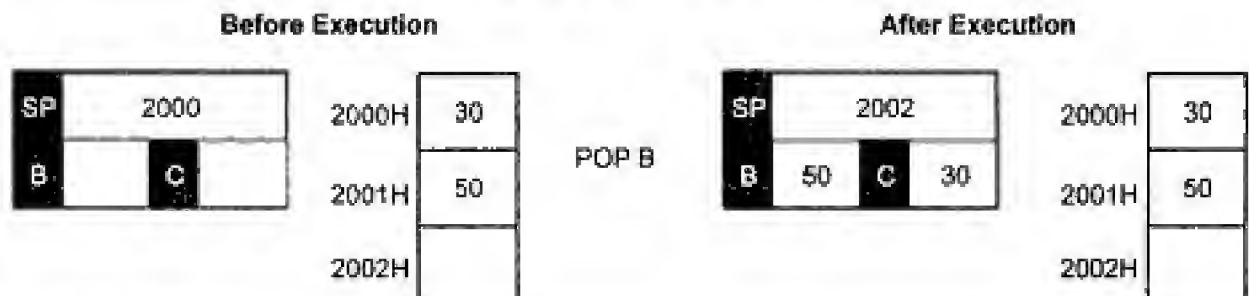


Fig. 3.6 Steps Involved in POP Operation

Example : SP = 2000H, (2000H) = 30H, (2001H) = 50H

POP B ; This instruction will copy the contents of memory location pointed by stack pointer, 2000H (i.e. data 30H) into the C register. It will then increment the stack pointer by one, 2001H and will copy the contents of memory location pointed by stack pointer, 2001H

(i.e. data 50H) into B register, and increment the stack pointer again by one.



4. POP PSW

This instruction copies the contents of memory location pointed by the stack pointer into the flag register and increments the stack pointer by one. It then copies the contents of memory location pointed by stack pointer into the accumulator and increments the stack pointer again by one.

Operation : Flag register \leftarrow (SP)

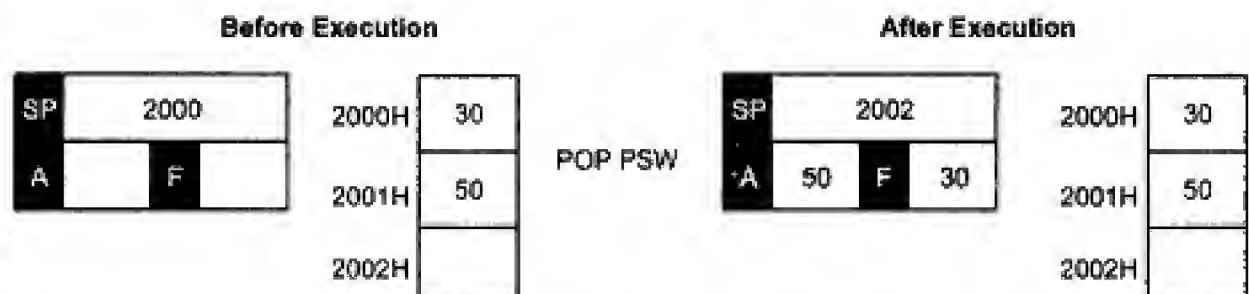
$SP \leftarrow SP + 1$

$A \leftarrow$ (SP)

$SP \leftarrow SP + 1$

Example : SP = 2000H, (2000H) = 30H, (2001H) = 50H

POP PSW : This instruction will copy the contents of memory location pointed by the stack pointer, 2000H (i.e. data 30H) into the flag register. It will then increment the stack pointer by one, 2001H and will copy the contents of memory location pointed by stack pointer into the accumulator and increment the stack pointer again by one.



5. SPHL

This instruction copies the contents of HL register pair into the stack pointer. The contents of H register are copied to higher order byte of stack pointer and contents of L register are copied to the lower byte of stack pointer.

Operation : $SP \leftarrow HL$

Example : $HL = 2500H$

SPHL ; This instruction will copy 2500H into stack pointer. So after execution of instruction stack pointer contents will be 2500H.

6. XTHL

This instruction exchanges the contents of memory location pointed by the stack pointer with the contents of L register and the contents of the next memory location with the contents of H register. This instruction does not modify stack pointer contents.

Operation : $L \leftrightarrow (SP)$
 $H \leftrightarrow (SP + 1)$

Example : $HL = 3040H$ and $SP = 2700H$, $(2700H) = 50H$, $(2701H) = 60H$

XTHL This instruction will exchange the contents of L register (40H) with the contents of memory location 2700H (i.e. 50H) and the contents of H register (30H) with the contents of memory location 2701H (i.e. 60H).

Before Execution

| | | | | | | |
|----|------|----|---|----|-------|----|
| SP | 2700 | | | | 2700H | 50 |
| | H | 30 | L | 40 | 2701H | 60 |
| | | | | | 2702H | |

XTHL

After Execution

| | | | | | | |
|----|------|----|---|----|-------|----|
| SP | 2700 | | | | 2700H | 40 |
| | H | 60 | L | 50 | 2701H | 30 |
| | | | | | 2702H | |

3.4.6 Branch Group

1. **JMP addr** This instruction loads the PC with the address given within the instruction and resumes the program execution from this location.

Operation : $PC \leftarrow \text{addr}$

Example :

JMP 2000H ; This instruction will load PC with 2000H and processor will fetch next instruction from this address.

2. **Jcond addr** This instruction causes a jump to an address given in the instruction if the desired condition occurs in the program before the execution of the instruction. The Table 3.3 shows the possible conditions for jumps.

| Instruction code | Description | Condition for jump |
|------------------|---------------------|--------------------|
| JC | Jump on carry | CY = 1 |
| JNC | Jump on not carry | CY = 0 |
| JP | Jump on positive | S = 0 |
| JM | Jump on minus | S = 1 |
| JPE | Jump on parity even | P = 1 |
| JPO | Jump on parity odd | P = 0 |
| JZ | Jump on zero | Z = 1 |
| JNZ | Jump on not zero | Z = 0 |

Table 3.3 Conditional jumps

Example : Carry flag = 1

JC 2000H : This instruction will cause a jump to an address 2000H i.e. program counter will load with 2000H since CF = 1.

3. CALL addr

A subroutine is a group of instructions, performs a particular subtask which is executed number of times. It is written separately. The microprocessor executes this subroutine by transferring program control to the subroutine program. After completion of subroutine program execution, the program control is returned back to the main program.

The CALL instruction is used to transfer program control to a subprogram or subroutine. This instruction pushes the current PC contents onto the stack and loads the given address into the PC. Thus the program control is transferred to the address given in the instruction. Stack pointer is decremented by two.

When the subroutine is called, the program control is transferred from calling program to the subroutine. After execution of subroutine it is necessary to transfer program control back to the calling program. To do this processor must remember the address of the instruction next to the CALL instruction. Processor saves this address on the stack when the CALL instruction is executed.

Note : The stack is a part of read/write memory set aside for storing intermediate results and addresses.

Operation : $(SP - 1) \leftarrow PC_H$
 $(SP - 2) \leftarrow PC_L$
 $SP \leftarrow SP - 2$
 $PC \leftarrow \text{addr}$

Example : Stack pointer = 3000H.

6000H CALL 2000H ; This instruction will store the address of instruction next to CALL (i.e. 6003H) on the stack and load PC

6003H — with 2000H.

Before Execution

| | |
|----|------|
| SP | 3000 |
| PC | 6000 |

2FFEh

2FFFh

3000H

CALL 2000H

After Execution

| | |
|----|------|
| SP | 2FFE |
| PC | 2000 |

2FFEh

2FFFh

3000H

03

60

4. C cond addr

This instruction calls the subroutine at the given address if a specified condition is satisfied. Before call it stores the address of instruction next to the call on the stack and decrements stack pointer by two. The Table 3.4 shows the possible conditions for calls.

| Instruction code | Description | Condition for CALL |
|------------------|---------------------|--------------------|
| CC | Call on carry | CY = 1 |
| CNC | Call on not carry | CY = 0 |
| CP | Call on positive | S = 0 |
| CM | Call on minus | S = 1 |
| CPE | Call on parity even | P = 1 |
| CPO | Call on parity odd | P = 0 |
| CZ | Call on zero | Z = 1 |
| CNZ | Call on not zero | Z = 0 |

Table 3.4 Conditional calls

Operation : If condition true $(SP - 1) \leftarrow PCH$
 $(SP - 2) \leftarrow PCL$
 $PC \leftarrow \text{addr}$
 else $PC \leftarrow PC + 3$

Example : Carry flag = 1, stack pointer = 4000H.

2000H CC 3000H ; This instruction will store the address of the next instruction i.e. 2003H on the stack and load the program counter with 3000H, since the carry flag is set.

5. RET

This instruction pops the return addr (address of the instruction next to CALL in the main program) from the stack and loads program counter with this return address. Thus transfers program control to the instruction next to CALL in the main program.

Operation : $PC_L \leftarrow (SP)$
 $PC_H \leftarrow (SP+1)$
 $SP \leftarrow SP + 2$

Example : If $SP = 27FDH$ and contents on the stack are as shown then

| | | |
|------|------|----|
| SP → | 27FD | 00 |
| | 27FE | 62 |
| | 27FF | |

RET : This instruction will load PC with 6200H and it will transfer program control to the address 6200H. It will also increment the stack pointer by two.

Before Execution

| | | | |
|----|------|-------|----|
| SP | 27FD | 27FDH | 00 |
| PC | | 27FEH | 62 |
| | | 27FFH | |

After Execution

| | | | |
|----|------|-------|----|
| SP | 27FF | 27FDH | 00 |
| PC | 6200 | 27FEH | 62 |
| | | 27FFH | |

RET

6. R condition

This instruction returns the control to the main program if the specified condition is satisfied. Table 3.5 shows the possible conditions for return.

| Instruction code | Description | Condition for RET |
|------------------|-----------------------|-------------------|
| RC | Return on carry | CY = 1 |
| RNC | Return on not carry | CY = 0 |
| RP | Return on positive | S = 0 |
| RM | Return on minus | S = 1 |
| RPE | Return on parity even | P = 1 |
| RPO | Return on parity odd | P = 0 |
| RZ | Return on zero | Z = 1 |
| RNZ | Return on not zero | Z = 0 |

Table 3.5 Conditions for return

7. PCHL

This instruction loads the contents of HL register pair into the program counter. Thus the program control is transferred to the location whose address is in HL register pair.

Operation : $PC \leftarrow HL$

Example : $HL = 6000H$

PCHL : This instruction will load 6000H into the program counter

8. RST n

This instruction transfers the program control to the specific memory address as shown in Table 3.6. This instruction is like a fixed address CALL instruction. These fixed addresses are also referred to as vector addresses. The processor multiplies the RST number by 8 to calculate these vector addresses. Before transferring the program control to the instruction following the vector address RST instruction saves the current program counter contents on the stack like CALL instruction

| Instruction code | Vector address |
|------------------|----------------------|
| RST 0 | $0 \times 8 = 0000H$ |
| RST 1 | $1 \times 8 = 0008H$ |
| RST 2 | $2 \times 8 = 0010H$ |
| RST 3 | $3 \times 8 = 0018H$ |
| RST 4 | $4 \times 8 = 0020H$ |
| RST 5 | $5 \times 8 = 0028H$ |
| RST 6 | $6 \times 8 = 0030H$ |
| RST 7 | $7 \times 8 = 0038H$ |

Table 3.6 Vector addresses for return instructions

Operation : $(SP - 1) \leftarrow PC_H$
 $(SP - 2) \leftarrow PC_L$
 $SP \leftarrow SP - 2, PC \leftarrow (n \times 8)$ in hex

Example : $SP = 3000H$

2000H RST 6; This instruction will save the current contents of the program counter (i.e. address of next instruction 2001H) on the stack and it will load the program counter with vector address $6 \times 8 = 48_{10} = 30H$ 0030H.

3.4.7 Input/Output

1. **IN addr(8-bit)** This instruction copies the data at the port whose address is specified in the instruction into the accumulator.

Operation : $A \leftarrow (addr)$

Example : Port address = 80H, data stored at port address 80H, (80H) = 10H

IN 80H ; This instruction will copy the data stored at address 80H, i.e. data 10H in the accumulator.

2. **OUT addr(8-bit)** This instruction sends the contents of accumulator to the output port whose address is specified within the instruction.

Operation : $(addr) \leftarrow A$

Example : $A = 40H$

OUT 50H ; This instruction will send the contents of accumulator(40H) to the output port whose address is 50H.

3.4.8 Machine Control Group

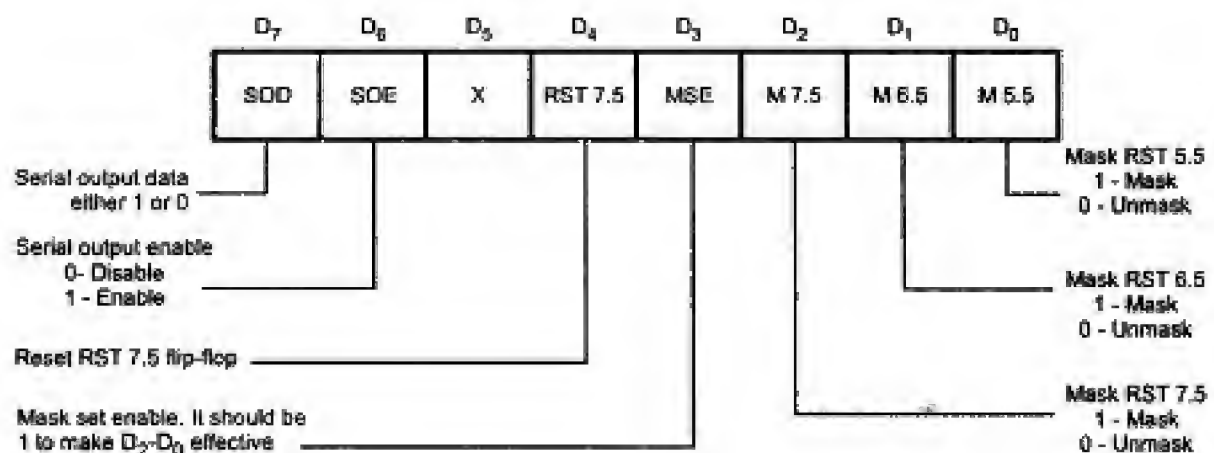
1. **EI** This instruction sets the interrupt enable flip flop to enable interrupts. When the microprocessor is reset or after interrupt acknowledge, the interrupt enable flip-flop is reset. This instruction is used to reenable the interrupts.

Operation : $IE (F/F) \leftarrow 1$

2. **DI** This instruction resets the interrupt enable flip-flop to disable interrupts. This instruction disables all interrupts except TRAP since TRAP is non-maskable interrupt (cannot be disabled. It is always enabled).

Operation : $IE (F/F) \leftarrow 0$

3. **NOP** No operation is performed.
4. **HLT** This instruction halts the processor. It can be restarted by a valid interrupt or by applying a RESET signal.
5. **SIM** This instruction masks the interrupts as desired. It also sends out serial data through the SOD pin. For this instruction command byte must be loaded in the accumulator.

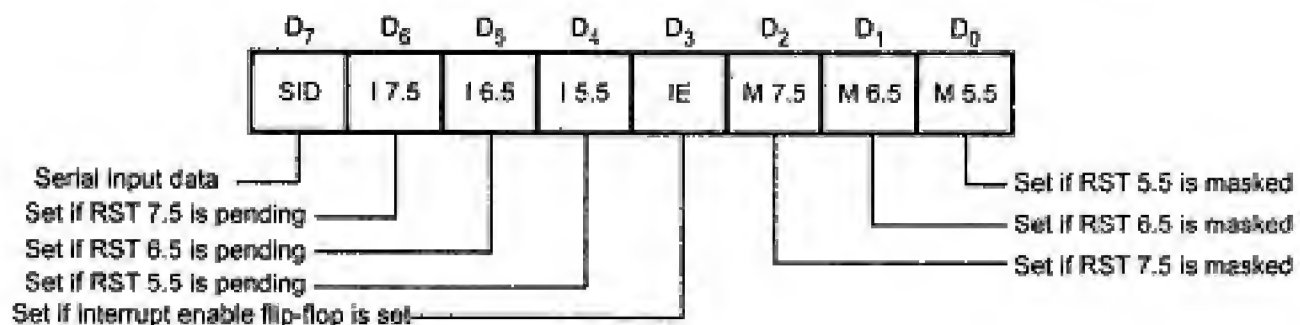


Example : ; i) A = 0EH

| D ₇ | D ₆ | D ₅ | D ₄ | D ₃ | D ₂ | D ₁ | D ₀ | Register A |
|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|------------|
| SOD | SOE | X | RST7.5 | MSE | M7.5 | M6.5 | M5.5 | |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0EH |

SIM ; This instruction will mask RST 7.5 and RST 6.5 interrupts where as RST 5.5 interrupt will be unmasked. It will also disable serial output.

6. RIM This instruction copies the status of the interrupts into the accumulator. It also reads the serial data through the SID pin.



Example :

RIM ; After execution of RIM instruction if the contents of accumulator are 4BH then we get following information.

| D ₇ | D ₆ | D ₅ | D ₄ | D ₃ | D ₂ | D ₁ | D ₀ | Register A |
|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|------------|
| SID | I 7.5 | I 6.5 | I 5.5 | IE | M 7.5 | M 6.5 | M 5.5 | |
| 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 4BH |

- i.e.
- a) RST 7.5 is pending
 - b) RST 5.5 and RST 6.5 are masked
 - c) Interrupt Enable flip-flop is set
 - d) Serial i/p data is zero.

3.5 Addressing Modes

Part of the programming flexibility for each microprocessor is the number and different kind of ways the programmer can refer to data stored in the memory. The different ways that a microprocessor can access data are referred to as addressing modes. The 8085 has 5 addressing modes. These are :

1. Immediate
2. Register
3. Direct
4. Indirect
5. Implied

1. Immediate Addressing Mode :

In an immediate addressing mode, 8 or 16 bit data can be specified as a part of instruction. In 8085, the instructions having 'I' letter fall under this category. 'I' indicates immediate addressing mode.

Example :

```
MVI A, 20H      ; Moves 8-bit immediate data (20H) into
                  ; accumulator
MVI M, 30H      ; Moves 8-bit immediate data (30H) into the
                  ; memory location pointed by HL register pair.
LXI SP, 2700H   ; Moves 16-bit immediate data (2700H) into SP.
LXI D, 10FFH    ; Moves 16-bit immediate data (10FFH) into DE
                  ; register pair ( D = 10H and E = FFH).
```

2. Register Addressing Mode :

The register addressing mode specifies the source operand, destination operand, or both to be contained in an 8085 registers. This results in faster execution, since it is not necessary to access memory locations for operands.

Example :

```
MOV A, B        ; Moves the contents of register B into the
                  ; accumulator.
SPHL            ; Moves the contents of HL register pair into
                  ; stack pointer.
ADD C            ; Adds the contents of register C into the
                  ; contents of accumulator and stores result in
                  ; the accumulator.
```

3. Direct Addressing Mode :

The direct addressing mode specifies the 16 bit address of the operand within the instruction itself. The second and third bytes of instruction contain this 16 bit address.

Example :

```
LDA 2000H       ; Loads the 8-bit contents of memory location
                  ; 2000H into the accumulator.
SHLD 3000H      ; Stores the HL register pair into two consecutive
                  ; memory locations. Lower byte i.e. the contents
                  ; of L register into memory location 3000H and
                  ; higher byte i.e. the contents of H register into
                  ; memory location 3001H.
```

4. Indirect Addressing Mode :

In indirect addressing mode, the memory address where the operand located is specified by the contents of a register pair.

Example :

```
LDAX B          ; Loads the accumulator with the contents of
                  ; memory location pointed by BC register pair.
MOV M, A        ; Stores the contents of accumulator into the
                  ; memory location pointed by HL register pair.
```

5. Implied Addressing Mode :

In implied addressing mode, opcode specifies the address of the operands.

Example :

```
CMA          ; Complements contents of accumulator.
RAL          ; Rotates the contents of accumulator left through
              ; carry.
```

Note : Many of the advanced processors support addressing mode called **index addressing mode**. In this mode, the address of the operand within the memory is generated by adding the offset/displacement to the register specified in the instruction. The offset/displacement is also a part of the instruction. In 8085 such addressing mode is not available. However, we can implement such kind of program structure, by using memory pointer (HL register), any other register pair and a instruction sequence given below :

```
LXI H, Base_addr      ; Loads the base address
LXI B, Offset/Displacement ; Loads the offset or displacement
DAD B                  ; Gives the addition of HL and BC
                        ; in HL register pair.
MOV A,M                ; Load the data from memory in the
                        ; accumulator
```

By incrementing or decrementing contents of BC register or loading another contents, we can change the index/offset/displacement.

3.6 Instruction Set Summary

| Data Transfer Group | | | | | |
|---------------------|-------------------|--|----------------|--------------|------------------------|
| Sr. no. | Instruction | Operation | Flags affected | No. of Bytes | Addressing mode |
| 1. | MVI r, data (8) | $r \leftarrow \text{data}(8)$ | No | 2 | Immediate |
| 2. | MVI M, data (8) | $(HL) \leftarrow \text{data}(8)$ | No | 2 | Immediate and Indirect |
| 3. | MOV rd, rs | $rd \leftarrow rs$ | No | 1 | Register |
| 4. | MOV M, rs | $(HL) \leftarrow rs$ | No | 1 | Indirect |
| 5. | MOV rd, M | $rd \leftarrow (HL)$ | No | 1 | Indirect |
| 6. | LXI rp, data (16) | $rp \leftarrow \text{data}(16)$ | No | 3 | Immediate |
| 7. | STA addr | $(addr) \leftarrow A$ | No | 3 | Direct |
| 8. | LDA addr | $A \leftarrow (addr)$ | No | 3 | Direct |
| 9. | SHLD addr | $(addr) \leftarrow L$ $(addr + 1) \leftarrow H$ | No | 3 | Direct |

| | | | | | |
|-----|-----------|---|----|---|----------|
| 10. | LHLD addr | $L \leftarrow (\text{addr})$ $HL \leftarrow (\text{addr}+1)$ | No | 3 | Direct |
| 11. | STAX rp | $(rp) \leftarrow A$ | No | 1 | Indirect |
| 12. | LDAX rp | $A \leftarrow (rp)$ | No | 1 | Indirect |
| 13. | XCHG | $H \leftrightarrow D, L \leftrightarrow E$ | No | 1 | Register |

Arithmetic Group

| Sr. no. | Instruction | Operation | Flags affected | No. of Bytes | Addressing mode |
|---------|-------------|---|----------------|--------------|-------------------|
| 1. | ADD r | $A \leftarrow A + r$ | All | 1 | Register |
| 2. | ADD M | $A \leftarrow A + (HL)$ | All | 1 | Register Indirect |
| 3. | ADI data(8) | $A \leftarrow A + \text{data}(8)$ | All | 2 | Immediate |
| 4. | ADC r | $A \leftarrow A + r + CY$ | All | 1 | Register |
| 5. | ADC M | $A \leftarrow A + (HL) + CY$ | All | 1 | Register indirect |
| 6. | ACI data(8) | $A \leftarrow A + \text{data}(8) + CY$ | All | 2 | Immediate |
| 7. | DAD rp | $HL \leftarrow HL + rp$ | CY | 1 | Register |
| 8. | Sub r | $A \leftarrow A - r$ | All | 1 | Register |
| 9. | Sub M | $A \leftarrow A - (HL)$ | All | 1 | Register Indirect |
| 10. | SUI data(8) | $A \leftarrow A - \text{data}(8)$ | All | 2 | Immediate |
| 11. | SBB r | $A \leftarrow A - r - CY$ | All | 1 | Register |
| 12. | SBB M | $A \leftarrow A - (HL) - CY$ | All | 1 | Register Indirect |
| 13. | SBI data | $A \leftarrow A - \text{data}(8) - CY$ | All | 2 | Immediate |
| 14. | DAA | $A(\text{BCD}) \leftarrow A(\text{Binary})$ | All | 1 | Implied |
| 15. | INR r | $r \leftarrow r + 1$ | S, Z, P, A, C | 1 | Register |
| 16. | INR M | $(HL) \leftarrow (HL) + 1$ | S, Z, P, A, C | 1 | Register Indirect |
| 17. | INX rp | $rp \leftarrow rp + 1$ | No | 1 | Register |
| 18. | DCR r | $r \leftarrow r - 1$ | S, Z, P, A, C | 1 | Register |
| 19. | DCR M | $(HL) \leftarrow (HL) - 1$ | S, Z, P, A, C | 1 | Register Indirect |
| 20. | DCX rp | $rp \leftarrow rp - 1$ | No | 1 | Register |

| Logic Group | | | | | |
|-------------|--------------|---|--------------------|---|-------------------|
| 1. | ANA r | $A \leftarrow A \wedge r$ | All, CY = 0 AC = 1 | 1 | Register |
| 2. | ANA M | $A \leftarrow A \wedge (HL)$ | All, CY = 0 AC = 1 | 1 | Register Indirect |
| 3. | ANI data(8) | $A \leftarrow A \wedge \text{data}(8)$ | All, CY = 0 AC = 1 | 2 | Immediate |
| 4. | XRA r | $A \leftarrow A \oplus r$ | All, CY = 0 AC = 0 | 1 | Register |
| 5. | XRA M | $A \leftarrow A \oplus (HL)$ | All, CY = 0 AC = 0 | 1 | Register Indirect |
| 6. | XRI data (8) | $A \leftarrow A \oplus \text{data} (8)$ | All, CY = 0 AC = 0 | 2 | Immediate |
| 7. | ORA r | $A \leftarrow A \vee r$ | All, CY = 0 AC = 0 | 1 | Register |
| 8. | ORA M | $A \leftarrow A \vee (HL)$ | All, CY = 0 AC = 0 | 1 | Register Indirect |
| 9. | ORI data (8) | $A \leftarrow A \vee \text{data} (8)$ | All, CY = 0 AC = 0 | 2 | Immediate |
| 10. | CMP r | $A - r$ | All | 1 | Register |
| 11. | CMP M | $A - (HL)$ | All | 1 | Register Indirect |
| 12. | CPI data (8) | $A - \text{data} (8)$ | All | 2 | Immediate |
| 13. | STC | $CY \leftarrow 1$ | CY | 1 | Implied |
| 14. | CMC | $CY \leftarrow \overline{CY}$ | CY | 1 | Implied |
| 15. | CMA | $A \leftarrow \overline{A}$ | No | 1 | Implied |

| Rotate Group | | | | | |
|--------------|-------------|---|----------------|--------------|-----------------|
| Sr. no. | Instruction | Operation | Flags affected | No. of Bytes | Addressing mode |
| 1. | RLC | $B_i + 1 \leftarrow B_i, B_0 \leftarrow B_7, CY \leftarrow B_7$ | CY | 1 | Implied |
| 2. | RRC | $B_i - 1 \leftarrow B_i, B_7 \leftarrow B_0, CY \leftarrow B_0$ | CY | 1 | Implied |
| 3. | RAL | $B_i + 1 \leftarrow B_i, B_0 \leftarrow CY, CY \leftarrow B_7$ | CY | 1 | Implied |
| 4. | RAR | $B_i - 1 \leftarrow B_i, B_7 \leftarrow CY, CY \leftarrow B_0$ | CY | 1 | Implied |
| Branch Group | | | | | |
| Sr. no. | Instruction | Operation | Flags affected | No. of Bytes | Addressing mode |
| 1. | JMP addr | $PC \leftarrow \text{addr}$ | No | 3 | Immediate |
| 2. | J cond addr | If condition true $PC \leftarrow \text{addr}$ | No | 3 | Immediate |
| 3. | PCHL | $PC \leftarrow HL$ | No | 1 | Register |

| | | | | | |
|----|----------------|--|----|---|-------------------|
| 4. | CALL addr | $(SP - 1) \leftarrow PC_H$ $(SP - 2) \leftarrow PC_L$ $SP \leftarrow SP - 2$ $PC \leftarrow \text{addr}$ | No | 3 | Immediate |
| 5. | CALL cond addr | If condition true $(SP - 1) \leftarrow PC_H$ $(SP - 2) \leftarrow PC_L$ $SP \leftarrow SP - 2$ $PC \leftarrow \text{addr}$ | No | 3 | Immediate |
| 6. | RET | $PC_L \leftarrow (SP)$ $PC_H \leftarrow (SP + 1)$ $SP \leftarrow SP + 2$ | No | 1 | Register Indirect |
| 7. | RET cond | If condition true $PC_L \leftarrow (SP)$ $PC_H \leftarrow (SP + 1)$ $SP \leftarrow SP + 2$ | No | 1 | Register Indirect |
| 8. | RST n | $(SP - 1) \leftarrow PC_H$ $(SP - 2) \leftarrow PC_L$ $SP \leftarrow SP - 2$ $PC \leftarrow (n \times 8) \text{ in hex}$ | No | 1 | Register Indirect |

Stack Group

| Sr. no. | Instruction | Operation | Flags affected | No. of Bytes | Addressing mode |
|---------|-------------|---|----------------|--------------|-------------------|
| 1. | PUSH rp | $SP \leftarrow SP - 1$ $(SP) \leftarrow rPH$ $SP \leftarrow SP - 1$ $(SP) \leftarrow rPL$ | No | 1 | Register Indirect |
| 2. | PUSH PSW | $SP \leftarrow SP - 1$ $(SP) \leftarrow A$ $SP \leftarrow SP - 1$ $(SP) \text{ flag register}$ | No | 1 | Register Indirect |
| 3. | POP rp | $rPL \leftarrow (SP)$ $SP \leftarrow SP + 1$ $rPH \leftarrow (SP)$ $SP \leftarrow SP + 1$ | No | 1 | Register Indirect |

| | | | | | |
|----|---------|--|----|---|-------------------|
| 4. | POP PSW | flag register \leftarrow (SP) SP \leftarrow SP + 1 A \leftarrow (SP) SP \leftarrow SP + 1 | No | 1 | Register Indirect |
| 5. | SPHL | SP \leftarrow HL | No | 1 | Register |
| 6. | XTHL | L \leftrightarrow (SP) H \leftrightarrow (SP + 1) | No | 1 | Register Indirect |

Input/Output Group

| Sr. no. | Instruction | Operation | Flags affected | No. of Bytes | Addressing mode |
|---------|--------------|-----------------------|----------------|--------------|-----------------|
| 1. | IN addr (8) | A \leftarrow (addr) | No | 2 | Direct |
| 2. | OUT addr (8) | (addr) \leftarrow A | No | 2 | Direct |

Machine Control Group

| Sr. no. | Instruction | Operation | Flags affected | No. of Bytes | Addressing mode |
|---------|-------------|------------------------|----------------|--------------|-----------------|
| 1. | EI | IE(F/F) \leftarrow 1 | No | 1 | — |
| 2. | DI | IE(F/F) \leftarrow 0 | No | 1 | — |
| 3. | NOP | No operation | No | 1 | — |
| 4. | HLT | Halts the processor | No | 1 | — |
| 5. | SIM | Serial interrupt mask | No | 1 | — |
| 6. | RIM | Read interrupt mask | No | 1 | — |

3.7 Assembly Language Programming

A program is a set of instructions arranged in the specific sequence to do the specific task. It tells the microprocessor what it has to do. The process of writing the set of instructions which tells the microprocessor what to do is called "Programming". In other words, we can say that programming is the process of telling the processor exactly how to solve a problem. To do this, the programmer must "speak" to the processor in a language which processor can understand.

3.7.1 Steps Involved in Programming

- **Specifying the problem :**
The first step in the programming is to find out which task is to be performed. This is called specifying the problem. If the programmer does not understand what is to be done, the programming process cannot begin.
- **Designing the problem-solution :**
During this process, the exact step by step process that is to be followed (program logic) is developed and written down.
- **Coding :**
Once the program is specified and designed, it can be implemented. Implementation begins with the process of coding the program. Coding the program means to tell the processor the exact step by step process in its language. Each processor has a set of instructions. Programmer has to choose appropriate instructions from the instruction set to build the program.
- **Debugging :**
Once the program or a part of program is coded, the next step is debugging the code. Debugging is the process of testing the code to see if it does the given task. If program is not working properly, debugging process helps in finding and correcting errors.

To write a program, programmer should know :

- How to develop program logic?
- How to tell the program to the processor?
- How to code the program?
- How to test the program?

3.7.2 Flowchart

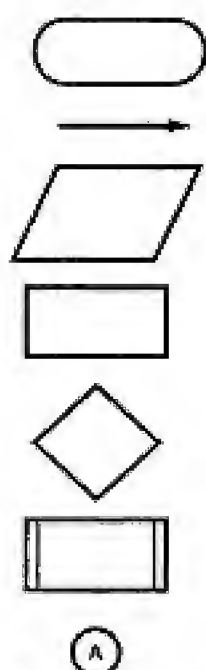


Fig. 3.7 Graphic symbols used in flowchart

To develop the programming logic programmer has to write down various actions which are to be performed in proper sequence. The flow chart is a graphical tool that allows programmer to represent various actions which are to be performed. The graphical representation is very useful for clear understanding of the programming logic.

The Fig. 3.7 shows the graphic symbols used in the flow chart.

Oval : It indicates start or stop operation.

Arrow : It indicates flow with direction.

Parallelogram : It indicates input/output operation.

Rectangle : It indicates process operation.

Diamond : It indicates decision making operation.

Double sided rectangle : It indicates execution of pre-defined process (subroutine).

Circle with alphabet : It indicates continuation.

A: Any alphabet

The Fig. 3.8 shows sample flowchart.

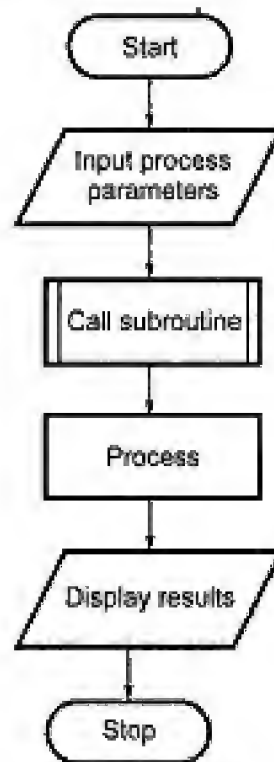


Fig. 3.8 Sample flowchart

3.7.3 Assembly Language Program

Let us define a program statement as 'write an assembly language program to add two numbers'. The three tasks are involved in this program :

- Load two hex numbers
- Add numbers and
- Store the result in the memory

These tasks can be symbolically presented as flowchart, as shown in the Fig. 3.9.

Next job is to find the suitable 8085 assembly language instruction/s for each task. These instructions are as follows :

Task 1 instructions :

```

MVI A, 20H    ; Load 20H as a first
               ; number in register A
MVI B, 40H    ; Load 40H as a second number
               ; in register B
  
```

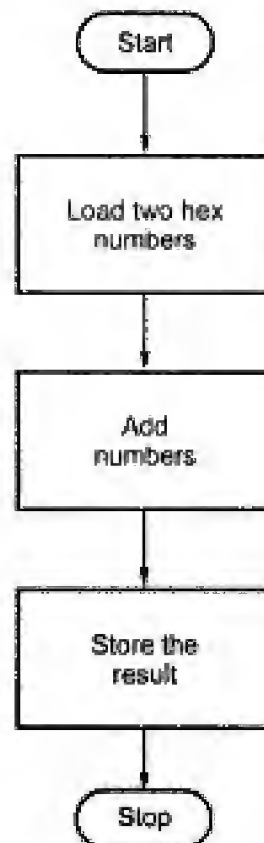



Fig. 3.9 Flowchart for addition of two numbers

Task 2 Instruction :

```

ADD B      ; Add two numbers and save
           ; result in register A
  
```

Task 3 Instruction :

```

STA 2200H  ; Store the result in memory
           ; location 2000H
HLT        ; Stop the program execution
  
```

We want to execute three tasks in a sequence, thus writing corresponding instructions in the same sequence constitutes an assembly language program.

3.7.4 Assembly Language Program to Machine Language Program

Once the assembly language program is ready, it is necessary to convert it in the machine language program. It is possible to do this by referring the proper hex code for each assembly instruction from the 8085 instruction set manual. This process is known as **hand assembly** and the resulted machine language program is also known as **hex code**. Let us see the hex code for our program.

| Mnemonics | Hex code |
|------------|---------------|
| MVI A, 20H | 3EH ← Opcode |
| | 20H ← Operand |

| | |
|------------|--|
| MVI B, 40H | 06H ← Opcode |
| | 40H ← Operand |
| ADD B | 80H ← Opcode |
| STA 2200H | 32H ← Opcode |
| | 00H ← Operand (lower byte of address) |
| | 22H ← Operand (higher byte of address) |
| HLT | 76H ← Opcode |

3.7.5 Storing Hex Code in the Memory

Once the hex code is ready, it has to be loaded in the memory of specially designed microprocessor system (Microprocessor training kit) for execution. To perform this task we should know the address range of read/write memory in the system. Let us assume that the read/write memory ranges from 2000H to 22FFH. The microprocessor training kit has keypad to enter the hex code in the memory. It provides a special routines (monitor program) to enter a hex code byte by byte and execute the program. Typical steps for storing hex code in the memory from address from address 2000H are as follows :

1. Reset the microprocessor system by pressing the RESET key.
2. Enter into store mode by pressing SET key.
3. Enter the address of the memory 2000H, where the first hex code (starting address of the program) is to be stored using hex keys.
4. Enter the hex code using hex keys.
5. Increment the memory address by 1 using INC key.
6. Repeat steps 4 and 5 until the last hex code.

3.7.6 Executing the Program

The microprocessor training kit provides a procedure to execute the program. To activate the procedure we have to enter the starting address of the program (2000H in our example). To enter this address we have to go into execute mode by pressing GO key and enter the starting address using hex keys. Once the starting address is entered, the program can be executed by pressing EXECUTE key. The EXECUTE key procedure loads the starting address of our program, 2000H into the program counter and program control is transferred from monitor program to our program.

After this microprocessor reads one hex code at a time, and when it fetches the complete instruction, it executes that instruction. Then it fetches the next instruction and this process continues until the last instruction in the program is executed.

3.8 Programming Examples

Lab Experiment 1 : Store 8-bit data in memory.

Statement : Store the data byte 52H into memory location 2000H.

```

Program 1 :
    MVI A, 52H      ; Store 52H in the accumulator
    STA 2000H       ; Copy accumulator contents at address 2000H
    HLT             ; Terminate program execution

Program 2 :
    LXI H, 2000H    ; Load HL with 2000H
    MVI M, 52H      ; Store 52H in memory location pointed
                    ; by HL register pair (2000H)
    HLT             ; Terminate program execution

```

The result of both programs will be the same. In program 1 direct addressing instruction is used, whereas in program 2 indirect addressing instruction is used.

Lab Experiment 2 : Exchange the contents of memory locations.

Statement : Exchange the contents of memory locations 1000H and 2000H

```

Program 1 :
    LDA 1000H       ; Get the contents of memory location 1000H
                    ; into accumulator
    MOV B, A        ; save the contents in B register
    LDA 2000H       ; Get the contents of memory location
                    ; 2000H into accumulator.
    STA 1000H       ; Store the contents of accumulator at
                    ; address 1000H.
    MOV A, B        ; Get the saved contents back into A
                    ; register
    STA 2000H       ; Store the contents of accumulator at
                    ; address 2000H
    HLT             ; Terminate program execution

Program 2 :
    LXI H, 1000H    ; Initialize HL register pair as a pointer
                    ; to memory location 1000H
    LXI D, 2000H    ; Initialize DE register pair as a pointer
                    ; to memory location 2000H
    MOV B, M        ; Get the contents of memory location
                    ; 1000H into B register
    LDAX D          ; Get the contents of memory location
                    ; 2000H into A register
    MOV M, A        ; Store the contents of A register into
                    ; memory location 1000H
    MOV A, B        ; Copy the contents of B register into
                    ; accumulator
    STAX D          ; Store the contents of A register into
                    ; memory location 2000H.
    HLT             ; Terminate program execution

```

In Program 1 direct addressing instructions are used, whereas in Program 2 indirect addressing instructions are used.

Lab Experiment 3 : Add two 8-bit numbers.

Statement : Add the contents of memory locations 2000H and 2001H and place the result in memory location 2002H.

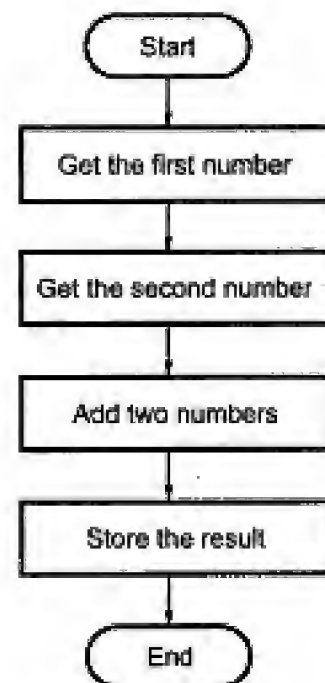
Sample problem

(2000H) = 14H
(2001H) = 89H
Result = 14H + 89H = 9DH

Source program

```
LXI H, 2000H ; HL points 2000H
MOV A, M      ; Get first operand
INX H         ; HL points 2001H
ADD M         ; Add second operand
INX H         ; HL points 2002H
MOV M, A      ; Store result at 2002H
HLT           ; Terminate program execution
```

Flowchart



Lab Experiment 4 : Subtract two 8-bit numbers.

Statement : Subtract the contents of memory location 2001H from the memory location 2000H and place the result in memory location 2002H.

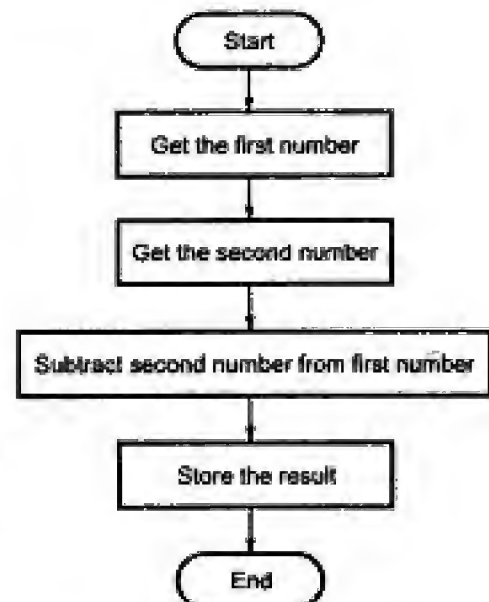
Sample problem

(2000H) = 51H
(2001H) = 19H
Result = 51H - 19H = 38H

Source program

```
LXI H, 2000H ; HL points 2000H
MOV A, M      ; Get first operand
INX H         ; HL points 2001H
SUB M         ; Subtract second operand
INX H         ; HL points 2002H
MOV M, A      ; Store result at 2002H
HLT           ; Terminate program execution
```

Flowchart

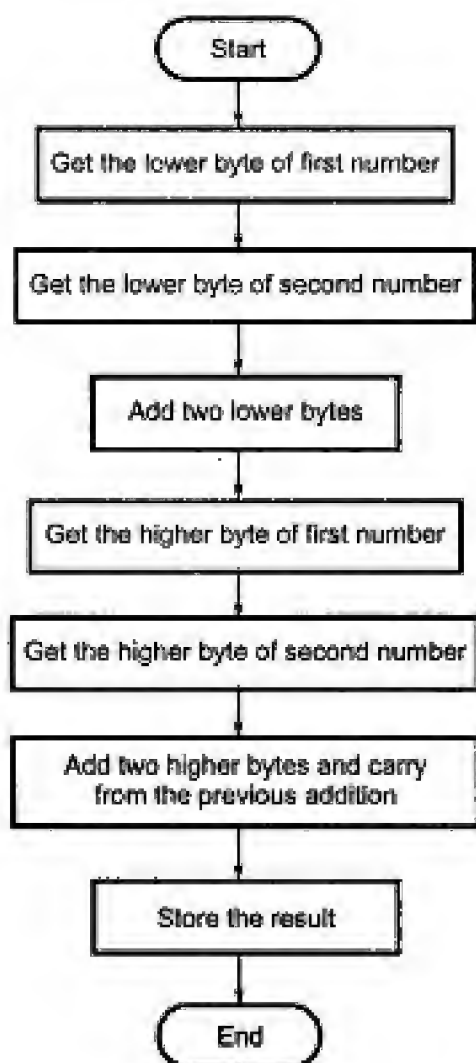


Lab Experiment 5 : Add two 16-bit numbers.

Statement : Add the 16-bit number in memory locations 2000H and 2001H to the 16-bit number in memory locations 2002H and 2003H. The most significant eight bits of the two numbers to be added are in memory locations 2001H and 2003H. Store the result in memory locations 2004H and 2005H with the most significant byte in memory location 2005H.

Sample problem

(2000H) = 15H
(2001H) = 1CH
(2002H) = B7H
(2003H) = 5AH
Result = 1C15 + 5AB7H = 76CCH
(2004H) = CCH
(2005H) = 76H

Flowchart

Source Program 1

```

LHLD 2000H    ; Get first 16-bit number in HL
XCHG          ; Save first 16-bit number in DE
LHLD 2002H    ; Get second 16-bit number in HL
MOV A, E      ; Get lower byte of the first number
ADD L         ; Add lower byte of the second number
MOV L, A      ; Store result in L register
MOV A, D      ; Get higher byte of the first number
ADC H         ; Add higher byte of the second number with
               ; carry
MOV H, A      ; Store result in H register
SHLD 2004H    ; Store 16-bit result in memory locations 2004H
               ; and 2005H.
HLT           ; Terminate program execution

```

Source program 2

```

LHLD 2000H    ; Get first 16-bit number
XCHG          ; Save first 16-bit number in DE
LHLD 2002H    ; Get second 16-bit number in HL
DAD D         ; Add DE and HL
SHLD 2004H    ; Store 16-bit result in memory locations 2004H
               ; and 2005H.
HLT           ; Terminate program execution

```

In program 1 eight bit addition instructions are used (ADD and ADC) and addition is performed in two steps. First lower byte addition using ADD instruction and then higher byte addition using ADC instruction. In program 2 16-bit addition instruction (DAD) is used.

Lab Experiment 6 : Subtract two 16-bit numbers.

Statement : Subtract the 16-bit number in memory locations 2002H and 2003H from the 16-bit number in memory locations 2000H and 2001H. The most significant eight bits of the two numbers are in memory locations 2001H and 2003H. Store the result in memory locations 2004H and 2005H with the most significant byte in memory location 2005H.

Sample problem

```

(2000H) = 19H
(2001H) = 6AH
(2002H) = 15H
(2003H) = 5CH
Result  = 6A19H - 5C15H = 0E04H
(2004H) = 04H
(2005H) = 0EH

```

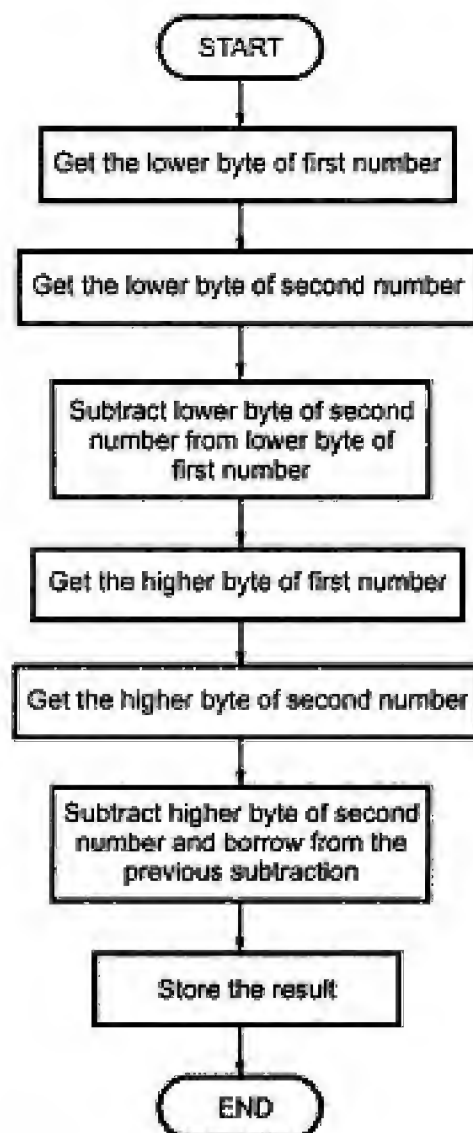
Source program

```

LHLD 2000H    ; Get first 16-bit number in HL
XCHG          ; Save first 16-bit number in DE
LHLD 2002H    ; Get second 16-bit number in HL
MOV A, E      ; Get lower byte of the first number
SUB L         ; Subtract lower byte of the second number
MOV L, A      ; Store the result in L register
MOV A, D      ; Get higher byte of the first number

```

```
SBB H      ; Subtract higher byte of second number with  
           ; borrow  
MOV H, A   ; Store 16-bit result in memory locations 2004H  
           ; and 2005H.  
SHLD 2004H ; Store 16-bit result in memory locations 2004H  
           ; and 2005H.  
HLT       ; Terminate program execution.
```

Flowchart**Lab Experiment 7 : Check results after execution of INR B, INR C and INX B instructions.**

Statement : If the contents of B = FFH and C = FFH then after execution of following instructions give the contents of register B and register C.

Instructions :

1. INR B
2. INR C
3. INX B

1. INR B

$$\begin{array}{r} B \rightarrow FFH \\ + \quad 01H \\ \hline 00H \rightarrow B \end{array}$$

$\therefore B = 00H$ and $C = FFH$

2. INR C

$$\begin{array}{r} C \rightarrow FFH \\ + \quad 01H \\ \hline 00H \rightarrow C \end{array}$$

$\therefore B = FFH$ and $C = 00H$

3. INX B

$$\begin{array}{r} BC \rightarrow \quad FF \quad FF \quad H \\ + \quad 00 \quad 01 \quad H \\ \hline 00 \quad 00 \quad H \rightarrow BC \end{array}$$

$\therefore B = 00H$ and $C = 00H$

Lab Experiment 8 : Check results after execution of DCR C, DCR B and DCX B instructions.

Statement : If the contents of $B = 00H$ and $C = 00H$ then after execution of following instructions give the contents of register B and register C.

Instructions :

1. DCR C
2. DCR B
3. DCX B

1. DCR C

$$\begin{array}{r} C \rightarrow 00H \\ - \quad 01H \\ \hline FFH \rightarrow C \end{array}$$

$\therefore B = 00H$ and $C = FFH$

2. DCR B

$$\begin{array}{r} B \rightarrow 00H \\ - \quad 01H \\ \hline FFH \rightarrow B \end{array}$$

$\therefore B = FFH$ and $C = 00H$

3. DCX B

$$\begin{array}{r} BC \rightarrow \quad 0 \quad 0 \quad 0 \quad 0 \quad H \\ - \quad 0 \quad 0 \quad 0 \quad 1 \quad H \\ \hline FF \quad FF \quad FF \quad FF \quad H \rightarrow BC \end{array}$$

$\therefore B = FFH$ and $C = FFH$

Lab Experiment 9 : Find the 1's complement of a number.

Statement : Find the 1's complement of the number stored at memory location 2200H and store the complemented number at memory location 2300H.

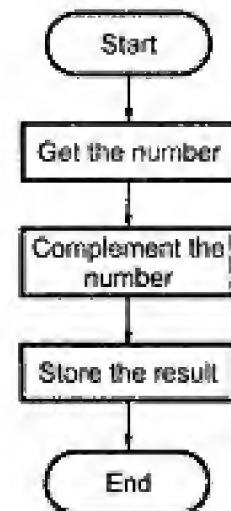
Sample problem

(2200H) = 55H

Result = (2300H) = AAH

Source program

```
LDA 2200H ; Get the number
CMA      ; Complement number
STA 2300H ; Store the result
HLT      ; Terminate program execution
```

**Lab Experiment 10 : Find the 2's complement of a number.**

Statement : Find the 2's complement of the number stored at memory location 2200H and store the complemented number at memory location 2300H.

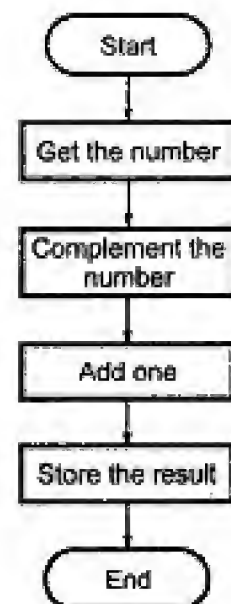
Sample problem

(2200H) = 55H

Result = (2300H) = AAH + 1 = ABH

Source program

```
LDA 2200 H ; Get the number
CMA       ; Complement the number
ADI, 01H  ; Add one in the number
STA 2300H ; Store the result
HLT       ; Terminate program execution
```

Flowchart

Lab Experiment 11 : Pack the two unpacked BCD numbers.

Statement : Pack the two unpacked BCD numbers stored in memory locations 2200H and 2201H and store result in memory location 2300H. Assume the least significant digit is stored at 2200H.

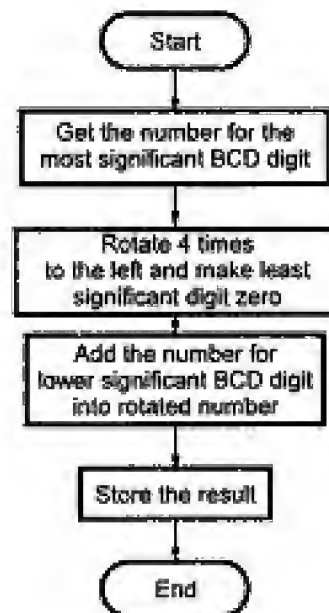
Flowchart**Sample problem**

(2200H) = 04
 (2201H) = 09
 Result = (2300H) = 94

Source program

```
LDA 2201H      ; Get the Most significant
                BCD digit

RLC
RLC
RLC
RLC            ; Adjust the position
ANI F0H        ; Make least significant BCD
                ; digit zero
MOV C,A        ; store the partial result
LDA 2200H      ; Get the lower BCD digit
ADD C          ; Add lower BCD digit
STA 2300H      ; Store the result
HLT            ; Terminate program execution
```

**Lab Experiment 12 : Unpack the BCD number.**

Statement : Two digit BCD number is stored in memory location 2200H. Unpack the BCD number and store the two digits in memory locations 2300H and 2301H such that memory location 2300H will have lower BCD digit.

Sample problem

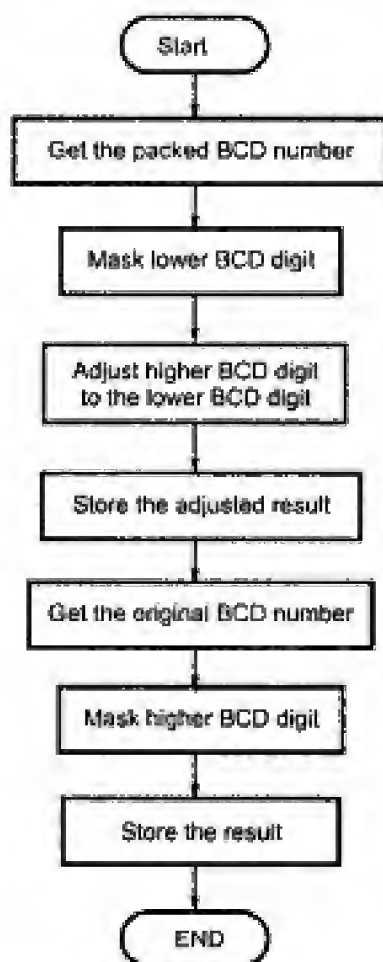
(2200H) = 58
 Result = (2300H) = 08 and (2301H) = 05

Flowchart

(See Fig. on next page)

Source program

```
LDA 2200H      ; Get the packed BCD number
ANI F0H       ; Mask lower nibble
RRC
RRC
RRC
RRC           ; Adjust higher BCD digit as a lower digit
STA 2301H     ; Store the partial result
LDA 2200H     ; Get the original BCD number
ANI 0FH      ; Mask higher nibble
STA 2201H     ; Store the result
HLT           ; Terminate program execution
```

Lab Experiment 13 : Sample subroutine program.

Statement : Read the program given below and state the contents of all registers after the execution of each instruction in sequence.

Main program :

```

6000H      LXI SP, 27FFH
6003H      LXI H, 2000H
6006H      LXI B, 1020H
6009H      CALL SUB
600CH      HLT
  
```

Subroutine program :

```

6100H  SUB : PUSH B
6101H      PUSH H
6102H      LXI B, 4080H
6105H      LXI H, 4090H
6108H      DAD B
6109H      SHLD 2200H
610CH      POP H
610DH      POP B
610EH      RET
  
```

Solution : The Table 3.7 shows the instruction sequence and the contents of all registers and stack after execution of each instruction.

| Sr. No. | Instructions | Registers contents in Hex | | | | | | | | | Memory locations (addresses are in Hex) |
|---------|--------------|---------------------------|----|----|---|---|----|----|------|------|--|
| | | A | B | C | D | E | H | L | SP | PC | |
| 1 | LXI SP,27FF | X | X | X | X | X | X | X | 27FF | 6003 | |
| 2 | LXI H,2000 | X | X | X | X | X | 20 | 00 | 27FF | 6006 | |
| 3 | LXI B,1020 | X | 10 | 20 | X | X | 20 | 00 | 27FF | 6009 | |
| 4 | CALL SUB | X | 10 | 20 | X | X | 20 | 00 | 27FD | 6100 | 27FE ← 60, 27FD ← 0C |
| 5 | PUSH B | X | 10 | 20 | X | X | 20 | 00 | 27FB | 6101 | 27FC ← 10, 27FB ← 20 |
| 6 | PUSH H | X | 10 | 20 | X | X | 20 | 00 | 27F9 | 6102 | 27FA ← 20, 27F9 ← 00 |
| 7 | LXI B,4080 | X | 40 | 80 | X | X | 20 | 00 | 27F9 | 6105 | |
| 8 | LXI H,4090 | X | 40 | 80 | X | X | 40 | 90 | 27F9 | 6108 | |
| 9 | DAD B | X | 40 | 80 | X | X | 81 | 10 | 27F9 | 6109 | |
| 10 | SHLD 2200 | X | 40 | 80 | X | X | 81 | 10 | 27F9 | 610C | 2200 ← 10, 2201 ← 81 |
| 11 | POP H | X | 40 | 80 | X | X | 20 | 00 | 27FB | 610D | |
| 12 | POP B | X | 10 | 20 | X | X | 20 | 00 | 27FD | 610E | |
| 13 | RET | X | 10 | 20 | X | X | 20 | 00 | 27FF | 600C | |
| 14 | HLT | X | 10 | 20 | X | X | 20 | 00 | 27FF | 600D | |

Table 3.7

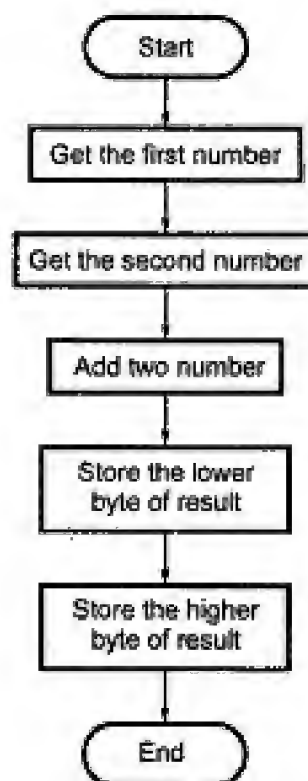
Lab Experiment 14 : Add contents of two memory locations.

Statement : Add the contents of memory locations 2000H and 2001H and place the result in the memory locations 2002H and 2003H.

Sample problem :

(2000H) = 7FH
 (2001H) = 89H
 Result = 7FH + 89H = 108H
 (2002H) = 08H
 (2003H) = 01H

Flowchart



Source program :

```

LXI H, 2000H ; HL Points 2000H
MOV A, M     ; Get first operand
INX H       ; HL Points 2001H
ADD M       ; Add second operand
INX H       ; HL Points 2002H
MOV M, A    ; Store the lower byte of result at 2002H
MVI A, 00H  ; Initialize higher byte result with 00H
ADC A       ; Add carry in the high byte result
INX H       ; HL Points 2003H
MOV M, A    ; Store the higher byte of result at 2003H
HLT         ; Terminate program execution
  
```

Lab Experiment 15 : Right shift data within the 8-bit register.

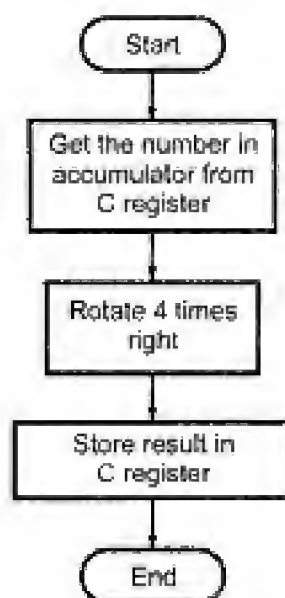
Statement : Write a program to shift an eight bit data four bits right. Assume that data is in register C.

Source program :

```

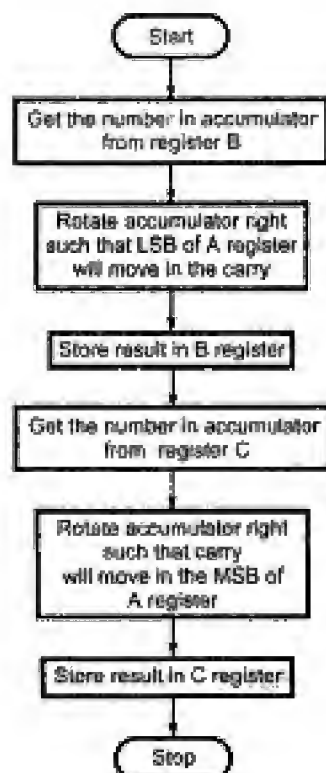
MOV A, C
RAR
RAR
RAR
RAR
MOV C, A
HLT
  
```

Flowchart

**Lab Experiment 16 : Right shift data within 16-bit register.**

Statement : Write a program to shift an 16-bit data 1-bit right. Assume data is in the BC register pair.

Flowchart



Source program :

```

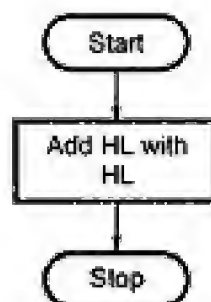
MOV  A,B
RAR
MOV  B,A
MOV  A,C
RAR
MOV  C,A
HLT

```

Lab Experiment 17 : Left shift 16-bit data within 16-bit register.

Statement : Write a program to shift an 16-bit data 1-bit left. Assume data is in the HL register pair.

Flowchart



Sample problem :

```

HL = 1025 = 0001 0000 0010 0101
∴   HL  = 0001 0000 0010 0101
+   HL  = 0001 0000 0010 0101
-----
Result  = 0010 0000 0100 1010

```

Source program :

```
DAD H
```

Lab Experiment 18 : Alter the contents of flag register.

Statement : Write a set of instructions to alter the contents of flag register in 8085.

Source Program :

```

PUSH PSW      ; Save flags on stack
POP  H        ; Retrieve flags in 'L'
MOV  A,L      ; Flags in accumulator
CMA          ; Complement accumulator
MOV  L,A      ; Accumulator in 'L'
PUSH H        ; Save on stack
POP  PSW      ; Back to flag register
HLT           ; Terminate program execution

```


Lab Experiment 19 : Find 2's complement of 16-bit number.

| | | |
|--------------|-------------------------|------------------------------------|
| Let | $x = 2000\text{ H}$ | $y = 4000\text{ H}$ |
| \therefore | $x + 1 = 2001\text{ H}$ | $\therefore y + 1 = 4001\text{ H}$ |

Source Program :

```

LDA 2000H      ; Load Accumulator with contents of
                ; 2000H location
CMA            ; complement the lower byte
ADD 01H        ; add 01 to the accumulator
STA 4000H      ; Store the accumulator data to 4000H
                ; location
LDA 2001H      ; Load accumulator with data from memory
                ; location 2001H
CMA            ; complement the higher byte
ADC 00H        ; add with carry to accumulator
HLT            ; Stop

```

Lab Experiment 20 : Simulation of CALL and RET instructions.

Statement : If CALL and RET instructions are not provided in the 8085, could it be possible to write subroutines for this microprocessor ? If so how will you call and return from the subroutine?

Solution : We know that,

1. CALL instruction transfers the program control to the subroutine program by loading the address of the subroutine program in the program counter, and before transferring program control it saves the address of the instruction after the CALL instruction on the stack.
2. RET instruction loads the program counter with return address from the stack and thus transfers the program control back to the instruction following the CALL.

Now our task is to implement the operation performed by these two instructions with the help of other instructions of 8085.

Let us implement RET instruction first. To implement this it is necessary to load program counter with return address. To load program counter with 16-bit address we have two instructions : JMP address and PCHL. But JMP address instruction can not be used for our purpose because JMP address loads program counter with fix address and the return address is not fix, it depends on, from where the subroutine is 'CALLED' in the main program. The other instruction, PCHL loads 16-bit contents of HL register pair into PC. If we manage to load return address into HL register pair, every time subroutine is called, then it is possible to implement RET instruction with the help of PCHL instruction. We can load the HL register pair with return address before we make a CALL for subroutine program.

The task to implement CALL instruction is simplified because we are going to store return address into the HL register pair before we make a 'CALL' for subroutine program.

Now it is only required to load program counter with the subroutine address. This can be implemented by executing JMP instruction. Here JMP instruction is suitable because subroutine starting address is a 'fix' address. The table shows how we can 'Call' and 'Return' from the subroutine without using CALL and RET instructions.

| Main program | Main program without CALL Instruction | | |
|---------------------|---------------------------------------|---------------|--|
| 6000H LXI SP, 27FFH | 6000H | LXI SP, 27FFH | ; Initialise stack pointer |
| | : | | |
| | : | | |
| 6010H CALL 2200H | 6010H | PUSH H | ; Saves HL register contents in stack, since HL is used for implementation of CALL and RET. After subroutine program execution the original HL contents are retrieved by executing POP H instruction |
| 6013H | 6011H | LXI H, 6018H | ; Loads HL with return address. |
| | 6014H | PUSH H | ; Stores return address in the stack |
| | 6015H | JMP 2200H | ; Loads program counter with 2200H |
| | 6018H | POP H | ; Loads HL with original contents of HL |
| | | | ; from the stack |

Table 3.8

| Subroutine program | Subroutine program without RET instruction | | |
|--------------------|--|-------|---|
| 2200 ... | 2200H | : | |
| | | : | |
| 2200H RET | 2220H | POP H | ; Loads HL with return address |
| | 2221H | PCHL | ; Loads program counter with return address |

Table 3.9

Lab Experiment 21 : Find the factorial of a number.

Statement : Write a program to calculate the factorial of a number between 0 to 8.

Solution : Main Program :

```

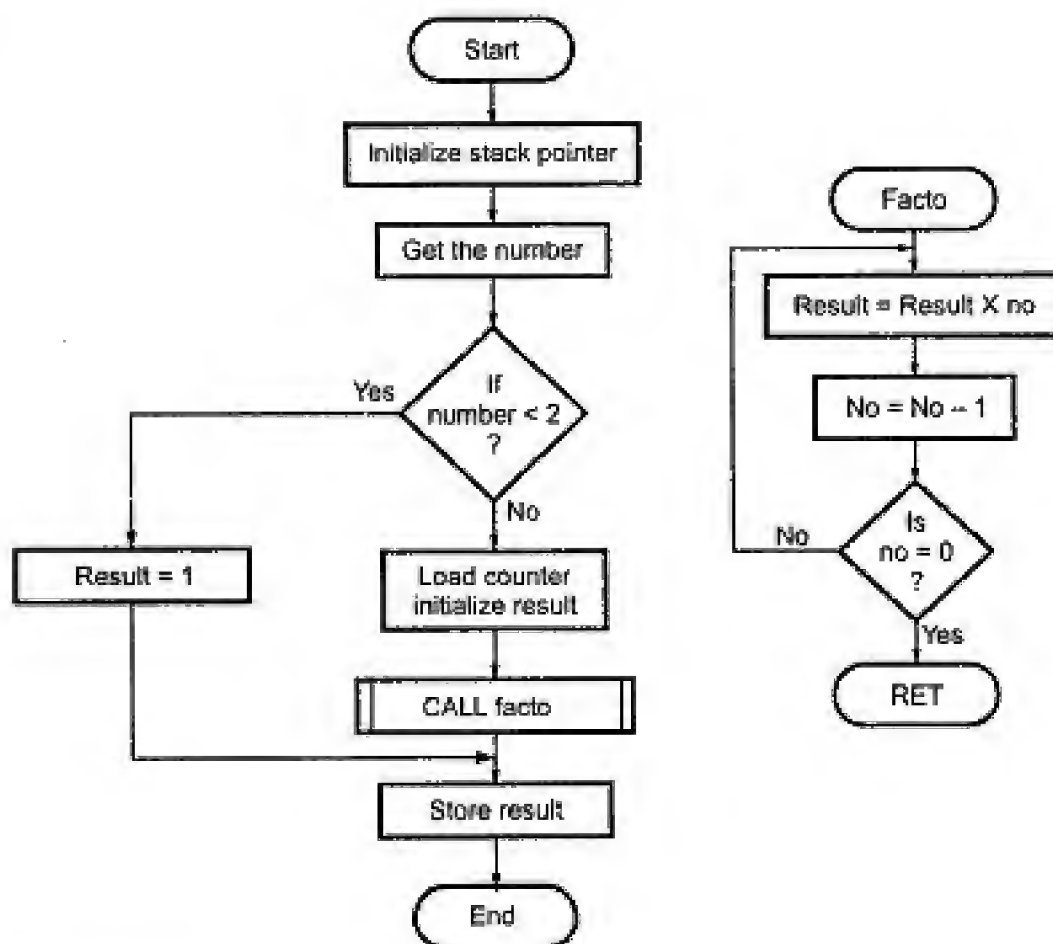
LXI SP, 27FFH      ; Initialize stack pointer
LDA 2200H           ; Get the number
CPI 02H             ; Check if number is greater than 1
JC LAST
MVI D,00H           ; Load number as a result
MOV E,A
DCR A
MOV C,A             ; Load counter one less than number

```

```

CALL FACTO      ; Call subroutine facto
XCHG            ; Get the result in HL
SHLD 2201H      ; Store result in the memory
JMP END
LAST : LXI H,0001H ; Store result = 01
SHLD 2201H      ;
END: HLT

```

Flowchart**Subroutine Program :**

```

FACTO : LXI H,0000H
MOV B,C      ; Load counter
BACK : DAD D  ;
DCR B        ;
JNZ BACK     ; Multiply by successive addition
XCHG         ; Store result in DE
DCR C        ; Decrement counter
CNZ FACTO    ; Call subroutine FACTO
RET          ; Return to main program

```

3.9 Instruction Comparisons

➡ **Example 3.1 :** Compare instructions *SUB A* and *MVI A, 00H*.

Solution :

| Parameter | SUB A | MVI A, 00H |
|-----------------------------|---|---|
| Number of instruction bytes | One byte instruction | Two byte instruction |
| Opcode | 97H | 3EH |
| Operation | This instruction subtracts the contents of register A from itself and stores result (00H) in register A | This instruction loads 00H in register A. |
| Addressing Mode | Register addressing mode | Immediate addressing mode |
| Flags | This instruction affects all flags. | This instruction does not affect flags |
| T-states Required | 4 | 7 |
| Machine Cycles Required | 1. Opcode fetch | 1. Opcode fetch 2. Memory read |

➡ **Example 3.2 :** Compare instructions *SUB B* and *CMP B*.

Solution :

| Parameter | SUB B | CMP B |
|-----------------------------|--|---|
| Number of instruction bytes | One byte instruction | One byte instruction |
| Opcode | 90H | B8H |
| Operation | This instruction subtracts the contents of register B from register A and stores result in register A and affects the flags according to contents of register A and B. | This instruction subtracts the contents of register B from register A and affects the flags according to contents of registers A and B. |
| Result | Result of $A - B$ is stored in register A | Result of $A - B$ is not stored in register A. The contents of register A are unchanged. |
| Addressing Mode | Register | Register |
| Flags | This instruction affects all flags | This instruction affects all flags |

| | | |
|--------------------------------|-----------------|-----------------|
| T-states Required | 4 | 4 |
| Machine Cycles Required | 1. Opcode fetch | 1. Opcode fetch |

➡ **Example 3.3 :** Compare instructions *LXI H, 2000H* and *LHLD 2000H*.

Solution :

| Parameter | LXI H, 2000H | LHLD 2000H |
|------------------------------------|---|--|
| Number of instruction bytes | Three byte instruction | Three byte instruction |
| Opcode | 21H | 2AH |
| Operand | 2000H | 2000H |
| Operation | This instruction loads 2000 H in the HL register pair | This instruction loads contents of 2000H memory location into L register and contents of 2001H memory location into H register |
| Addressing Mode | Immediate | Direct |
| Flags | This instruction does not affect flags | This instruction does not affect flags |
| Required T-States | 10 | 16 |
| Required Machine Cycles | 1. Opcode fetch 2. Memory read 3. Memory read | 1. Opcode fetch 2. Memory read 3. Memory read 4. Memory read 5. Memory read |

➡ **Example 3.4 :** Compare instruction *JMP 2000 H* and *PCHL*.

Solution :

| Parameter | JMP 2000H | PCHL |
|------------------------|------------------------|-------------------------|
| Number of bytes | Three byte instruction | One byte instruction |
| Opcode | C3H | E9H |
| Operand | 2000H | Contents of HL register |

| | | |
|--------------------------------|---|---|
| Operation | This instruction copies 2000H in PC so that processor fetches next instruction from address 2000H | This instruction copies the contents of HL register pair into PC so that processor fetches next instruction from address specified by HL register pair. |
| Addressing Mode | Direct | Indirect |
| Flags | This instruction does not affect flags | This instruction does not affect flags |
| Required T- states | 10 | 6 |
| Required Machine Cycles | 1. Opcode fetch 2. Memory read 3. Memory read | 1. Opcode fetch |

➡ **Example 3.5 :** Compare instructions *HLT* and *NOP*.

Solution :

| Parameter | HLT | NOP |
|-----------------------------------|---|--|
| Number of Instruction byte | One byte instruction. | One byte instruction. |
| Opcode | 76H. | 00H. |
| Operation | This instruction halts the processor. | No operation is performed. |
| Flags | This instruction does not affect flags. | This instruction does not affect flags. |
| Required T- states | T- states required are undefined because this instruction halts the processor. The processor can be restarted by a valid interrupt or by applying a RESET signal. | 4-T states. After this instruction processor fetches the next instruction after NOP. |

➡ **Example 3.6 :** List out differences and similarities between CALL-RET and PUSH-POP instructions.

Solution : Differences :

| Sr. No. | CALL- RET | PUSH- POP |
|---------|--|---|
| 1 | These instructions are used for the execution of subroutine | These instructions are used to store register data temporarily in memory. |
| 2 | CALL instruction store the address of next instruction after it in the stack and loads PC with address given in the instruction. | PUSH instruction stores register contents in the stack. |
| 3 | RET instruction loads the address from stack into PC. | POP instruction gets the register contents from the stack. |

Similarities :

1. They use stack memory.
2. CALL and PUSH instructions decrement stack pointer by 2.
3. RET and POP instructions increment stack pointer by 2.
4. Instructions do not affect flags except POP PSW instruction.

Review Questions

1. Explain the classification of the instruction set of 8085 microprocessor with suitable examples.
2. Explain various elements of an instructions.
3. Give the instruction formats for 8085.
4. Give the opcode formats for 8085.
5. Explain various data formats supported by 8085.
6. With the help of one example in each case explain the effect of the following instructions in 8085.

| | |
|---------------|------------|
| a. LHL D addr | b. ADD M |
| c. RST 4 | d. XTHL |
| e. DAA | f. CP 2000 |
| g. DAD B | h. IN 20H |
| i. RIM | j. SIM |
7. Write the two ways to initialize stack pointer at FFFFH.
8. Compare the following pairs of instructions with their opcodes, operations, instruction bytes, addressing modes, affected flags and the results.

| | | |
|---------------|-----|------------|
| a. MVI A, 00H | and | XRA A |
| b. SUB B | and | CMP B |
| c. JMP 2700 | and | PCHL |
| d. XTHL | and | SPHL |
| e. LDA 2000H | and | LHLD 2000H |
| f. RRC | and | RAR |

9. How many times will the two JNZ instructions be executed in the following sequence ? What will be the contents of H and L when program control reaches to HLT instruction?

```
                LXI H, 0503H
LOOP :          DCR L
                JNZ LOOP
                DCR H
                JNZ LOOP
                HLT
```

10. Explain the following instructions :

i) LHLD 8850 ii) XTHL iii) DAD H iv) INR M.

11. Analyse the 8085 below to reveal the arithmetic operation performed :

```
MVI A, 07
RLC
MOV B, A
RLC
RLC
RLC
ORA B
```

12. Write an 8085 code to obtain 2's complement on the 16 number stored at locations x and x+1. Store the result in y and y+1 locations.

13. Explain the contents of accumulator to run SIM instruction.

14. Explain restart as a software instruction. Explain the implementation of RST5.

15. Explain the instructions RIM and SIM.

16. Define the instruction and instruction set.

17. Explain the operational difference between the following pairs of instructions.

i) SPHL and XTHL ii) CALL addr and JMP addr
iii) LHLD and SHLD addr iv) XRA A and MVI A, 00H
v) INR A and ADI 01 H vi) DAD RP and DAA.

18. Explain the operation of following instructions and specify addressing mode and number of M/C required :

i) DAA ii) DAD B
iii) XTHL iv) CNC addr.

19. Write a program to unpacks a two digit BCD number stored at memory location 1C00H.

20. Write an 8085 ALP to subtract two BCD numbers using 10's complement arithmetic.

21. Explain the operation performed by 8085 when the following instructions are executed.

i) SBB C ii) RRC
iii) LDAX B iv) XTHL.

22. Is it possible to check AC flag status of 8085 ? Explain.

23. *With an ALP to clear all flags load the data byte FFH into the accumulator ; increment the accumulator, mask all the flags except the carry flag and display the carry flag at PORT 0. Again load the accumulator the data byte. FFH ; add 01H to it and display the carry flag at PORT 1. Explain the difference in the answer.*
24. *Explain the various steps involved while executing CALL instruction with an example.*
25. *Explain DAA instruction with example.*
26. *What are addressing modes used in the following instructions ? Explain*
MVI M, 03FH
IN 4
PUSH B
RET
27. *What is program ?*
28. *Give the steps involved in programming.*
29. *What is flowchart ? Explain its use.*
30. *Explain the process of writing assembly language program with the help of example.*
31. *What do you mean by hand assembly ? Explain with the help of example.*
32. *Explain the process of executing the program on the microprocessor training kit.*



Programming Techniques

In the last chapter we have seen the instruction set of 8085 and some simple assembly language programs using it. We know that, the program is an implementation of certain logic by executing group of instructions. To implement program logic we need to take help of some common programming techniques such as looping, counting, indexing and code conversion.

In this chapter, we are going to study how to implement these programming techniques using 8085 assembly language and some programming examples using them. This chapter also introduces the BCD arithmetic and programming techniques to implement BCD arithmetic using 8085 assembly language.

4.1 Looping, Counting and Indexing

Before going to implement these techniques, we will get conversant with these techniques and understand their use.

Looping : In this technique, the program is instructed to execute certain set of instructions repeatedly to execute a particular task number of times. For example, to add ten numbers stored in the consecutive memory locations we have to perform addition ten times.

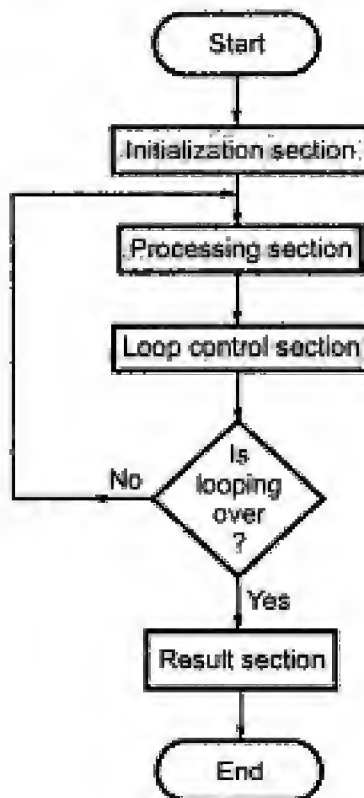
Counting : This technique allows programmer to count how many times the instruction/set of instructions are executed.

Indexing : This technique allows programmer to point or refer the data stored in sequential memory locations one by one. Let us see the program loop to understand looping, counting and indexing.

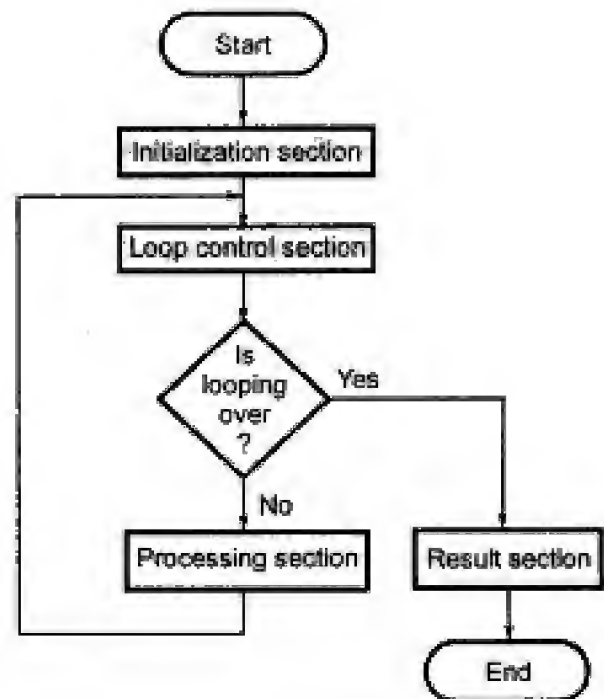
The program loop is the basic structure which forces the processor to repeat a sequence of instructions. Loops have four sections.

- | | |
|----------------------------|------------------------|
| 1. Initialization section. | 2. Processing section. |
| 3. Loop control section | 4. Result section. |

Flowchart



Flowchart 1



Flowchart 2

1. The initialization section establishes the starting values of
 - loop counters for counting how many times loop is executed,
 - address registers for indexing which give pointers to memory locations and
 - other variables
2. The actual data manipulation occurs in the processing section. This is the section which does the work.
3. The loop control section updates counters, indices (pointers) for the next iteration.
4. The result section analyzes and stores the results.

Note : The processor executes initialization section and result section only once, while it may execute processing section and loop control section many times. Thus, the execution time of the loop will be mainly dependent on the execution time of the processing section and loop control section. The flowchart 1 shows typical program loop. The processing section in this flowchart is always executed at least once. If you interchange the position of the processing and loop control section then it is possible that the processing section may not be executed at all, if necessary. Refer flowchart 2.

Program Examples

Lab Experiment 22 : Calculate the sum of series of numbers.

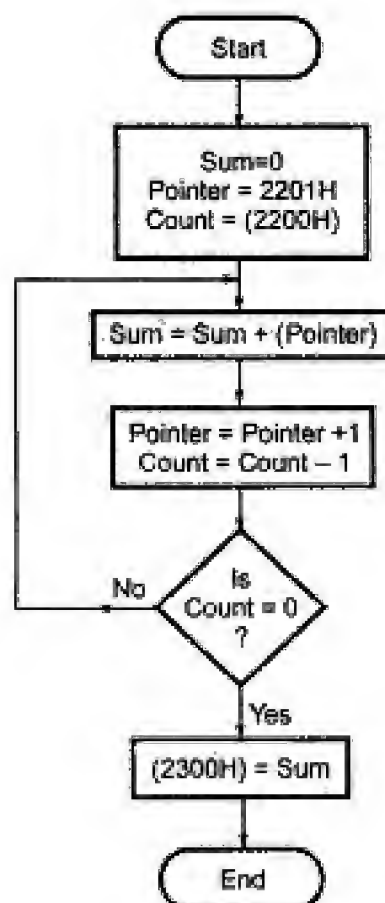
Statement : Calculate the sum of series of numbers. The length of the series is in memory location 2200H and the series itself begins from memory location 2201H.

- Assume the sum to be 8 bit number so you can ignore carries. Store the sum at memory location 2300H.
- Assume the sum to be 16 bit number. Store the sum at memory locations 2300H and 2301H.

a. Sample problem

| | | |
|---------|---|-------------------------|
| 2200H | = | 04H |
| 2201H | = | 20H |
| 2202H | = | 15H |
| 2203H | = | 13H |
| 2204H | = | 22H |
| Result | = | 20 + 15 + 13 + 22 = 6AH |
| ∴ 2300H | = | 6AH |

Flowchart



Source program

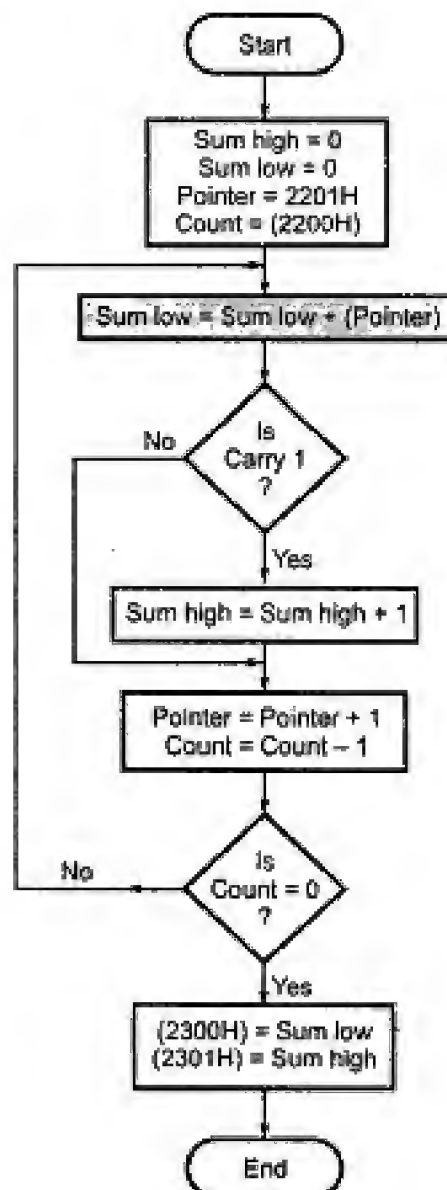
```

        LDA 2200H
        MOV C, A           ; Initialize counter
        SUB A              ; sum = 0
        LXI H, 2201H       ; Initialize pointer
BACK :   ADD M             ; SUM = SUM + data
        INX H             ; increment pointer
        DCR C             ; Decrement counter
        JNZ BACK          ; if counter  $\neq$  0 repeat
        STA 2300H         ; store sum
        HLT              ; Terminate program execution

```

b. Sample problem

$2200H = 04H$ $2201H = 9AH$
 $2202H = 52H$ $2203H = 89H$ $2204H = 3EH$
 $\text{Result} = 9AH + 52H + 89H + 3EH = 1B3H$
 $\therefore 2300H = B3H$ Lower byte $2301H = 01H$ Higher byte

Flowchart

Source program

```

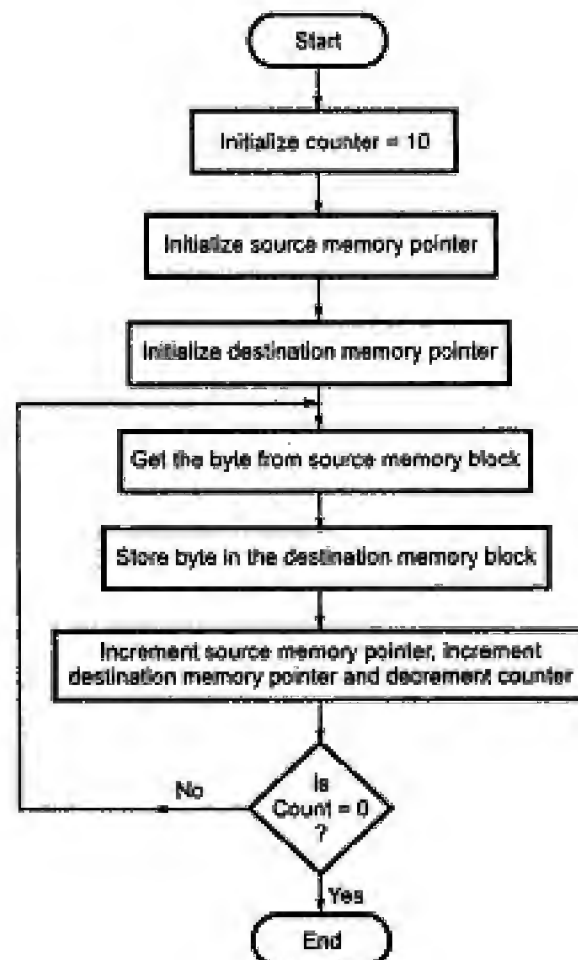
        LDA 2200H
        MOV C, A      ; Initialize counter
        LXI H, 2201H  ; Initialize pointer
        SUB A          ; Sumlow = 0
        MOV B, A      ; Sumhigh = 0
BACK :   ADD M          ; Sum = sum + data
        JNC SKIP
        INR B          ; Add carry to MSB of SUM
SKIP :   INX H          ; Increment pointer
        DCR C          ; Decrement counter
        JNZ BACK      ; Check if counter ≠ 0 repeat
        STA 2300H      ; Store lower byte
        MOV A, B
        STA 2301H      ; Store higher byte
        HLT           ; Terminate program execution

```

Lab Experiment 23 : Data transfer from memory block B1 to memory block B2.

Statement : Transfer ten bytes of data from one memory to **another** memory block. Source memory block starts from memory location 2200H where as **destination** memory block starts from memory location 2300H.

Flowchart



Source program

```

        MVI C, 0AH    ; Initialize counter
        LXI H, 2200H  ; Initialize source memory pointer
        LXI D, 2300H  ; Initialize destination memory pointer
BACK :   MOV A, M      ; Get byte from source memory block
        STAX D         ; Store byte in the destination memory
                        ; block
        INX H          ; Increment source memory pointer
        INX D          ; Increment destination memory pointer
        DCR C          ; Decrement counter
        JNZ BACK       ; If counter ≠ 0 repeat
        HLT           ; Terminate program execution

```

Lab Experiment 24 : Multiply two 8-bit numbers.

Statement : Multiply two 8-bit numbers stored in memory locations 2200H and 2201H. Store the result in memory locations 2300H and 2301H.

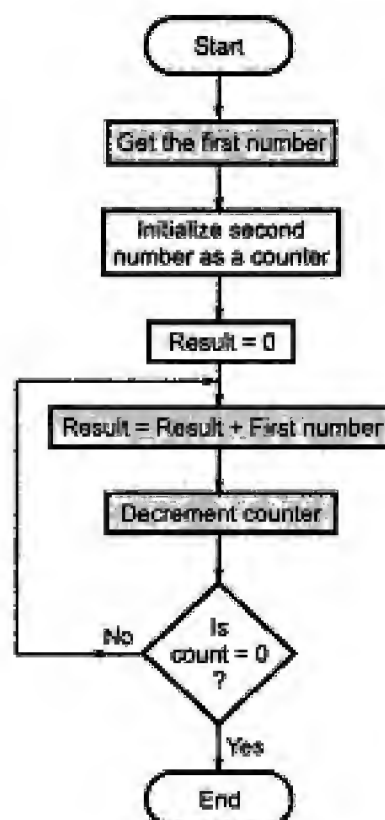
Sample problem

```

(2200H) = 03H
(2201H) = B2H
Result   = B2H + B2H + B2H
          = 216H
(2300H)  = 16H
(2301H)  = 02H

```

Note : In 8085 multiplication can be done by repetitive addition.

Flowchart

Source program

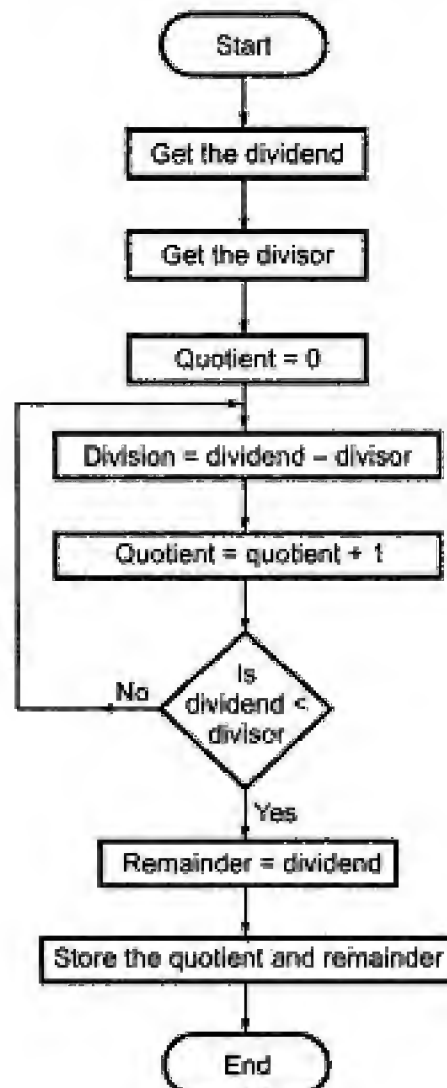
```

    LDA 2200H
    MOV E, A
    MVI D, 00      ; Get the first number in DE register pair
    LDA 2201H
    MOV C, A       ; Initialize counter
    LXI H, 0000H   ; Result = 0
BACK : DAD D       ; Result = result + first number
    DCR C          ; decrement count
    JNZ BACK       ; If count  $\neq$  0 repeat
    SHLD 2300H     ; Store result
    HLT           ; Terminate program execution

```

Lab Experiment 25 : Divide 16-bit number by 8-bit number.

Statement : Divide 16 bit number stored in memory locations 2200H and 2201H by the 8 bit number stored at memory location 2202H. Store the quotient in memory locations 2300H and 2301H and remainder in memory locations 2302H and 2303H.

Flowchart

Sample problem

(2200H) = 60H
 (2201H) = A0H
 (2202H) = 12H
 Result = A060H/12H = 8E8H Quotient
 and 10H remainder
 2300H) = E8H
 2301H) = 08H
 2302H) = 10H
 2303H) = 00H

Source program

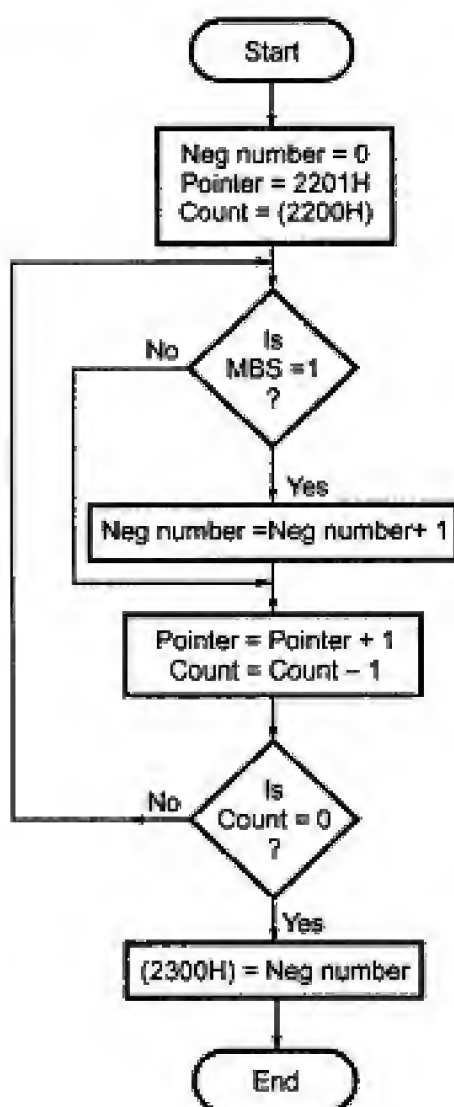
```

                LHLD 2200H    ; Get the dividend
                LDA 2202H
                MOV C, A      ; Get the divisor
                LXI D, 0000H  ; Quotient = 0
BACK :          MOV A, L
                SUB C         ; Subtract divisor
                MOV L, A      ; Save partial result
                JNC SKIP      ; if CY ≠ 1 jump
                DCR H         ; Subtract borrow of previous
                subtraction
SKIP :          INX D         ; Increment quotient
                MOV A, H
                CPI, 00       ; Check if dividend < divisor
                JNZ BACK      ; if no repeat
                MOV A, L
                CMP C
                JNC BACK
                SHLD 2302H    ; Store the remainder
                XCHG
                SHLD 2300H    ; Store the quotient
                HLT           ; Terminate program execution
  
```

Lab Experiment 26 : Find the negative numbers in a block of data.

Statement : Find the number of negative elements (most significant bit 1) in a block of data. The length of the block is in memory location 2200H and the block itself begins in memory location 2201H. Store the number of negative elements in memory location 2300H.

Flowchart



Sample problem

(2200H) = 04H

(2201H) = 56H

(2202H) = A9H

(2203H) = 73H

(2204H) = 62H

Result = 02 since 2202H and 2204H contain numbers with a MSB of 1.

Source program

```

LDA 2200H
MOV C,A      ; initialize count
MVI B, 00    ; Negative number = 0
LXI H, 2201H ; Initialize pointer
BACK : MOV A,M ; Get the number
      ANI 80H ; Check for MSB
      JZ SKIP ; If MSB = 1
  
```

```

      INR B      ; Increment negative number count
SKIP : INX H      ; Increment pointer
      DCR C      ; Decrement count
      JNZ BACK   ; If count  $\neq$  0 repeat
      MOV A, B    ;
      STA 2300H   ; Store the result
      HLT        ; Terminate program execution

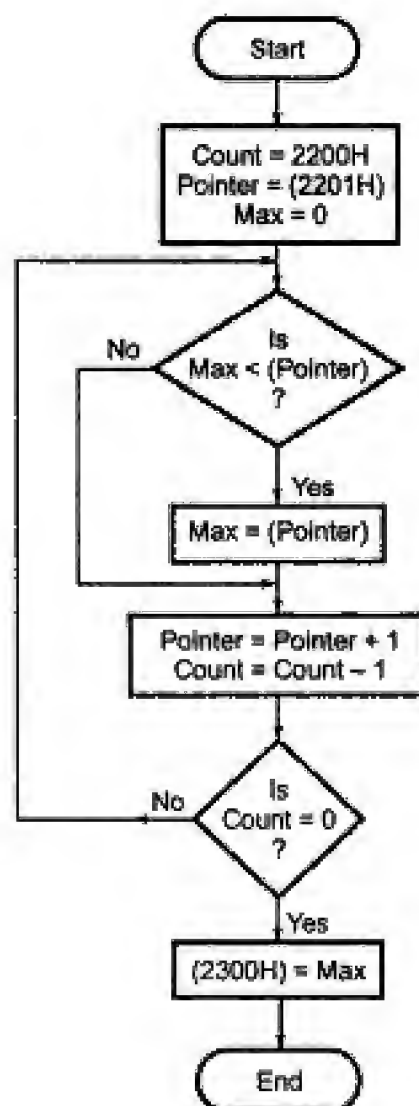
```

Lab Experiment 27 : Find the largest of given numbers.

Statement :

Find the largest number in a block of data. The length of the block is in memory location 2200H and the block itself start from memory location 2201H. Store the maximum number in memory location 2300H. Assume that the number in the block are all 8 bit unsigned binary numbers.

Flowchart



Sample problem

(2200H) = 04
 (2201H) = 34H
 (2202H) = A9H
 (2203H) = 78H
 (2204H) = 56H
 Result = (2202H) = A9H.

Source program

```

        LDA 2200H
        MOV C,A      ; Initialize counter
        XRA A        ; Maximum = Minimum possible value = 0
        LXI H, 2201H ; Initialize pointer
BACK :   CMP M        ; Is number > maximum
        JNC SKIP
        MOV A,M       ; Yes, replace maximum
SKIP :   INX H
        DCR C
        JNZ BACK
        STA 2300H     ; Store maximum number
        HLT           ; Terminate program execution
  
```

Lab Experiment 28 : Count number of one's in a number.

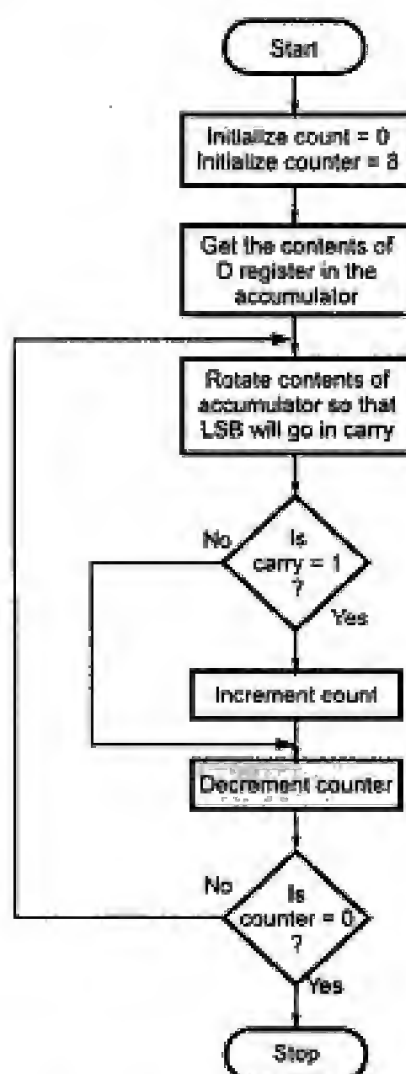
Statement : Write a program to count number of 1's in the contents of D register and store the count in the B register.

Source program :

```

        MVI B,00H
        MVI C,08H
        MOV A,D
BACK :   RAR
        JNC SKIP
        INR B
SKIP :   DCR C
        JNZ BACK
        HLT
  
```


Flowchart

**Lab Experiment 29 : Arrange numbers in the ascending order.**

Statement : Write a program to sort given 10 numbers from memory location 2200H in the ascending order.

Source program :

```

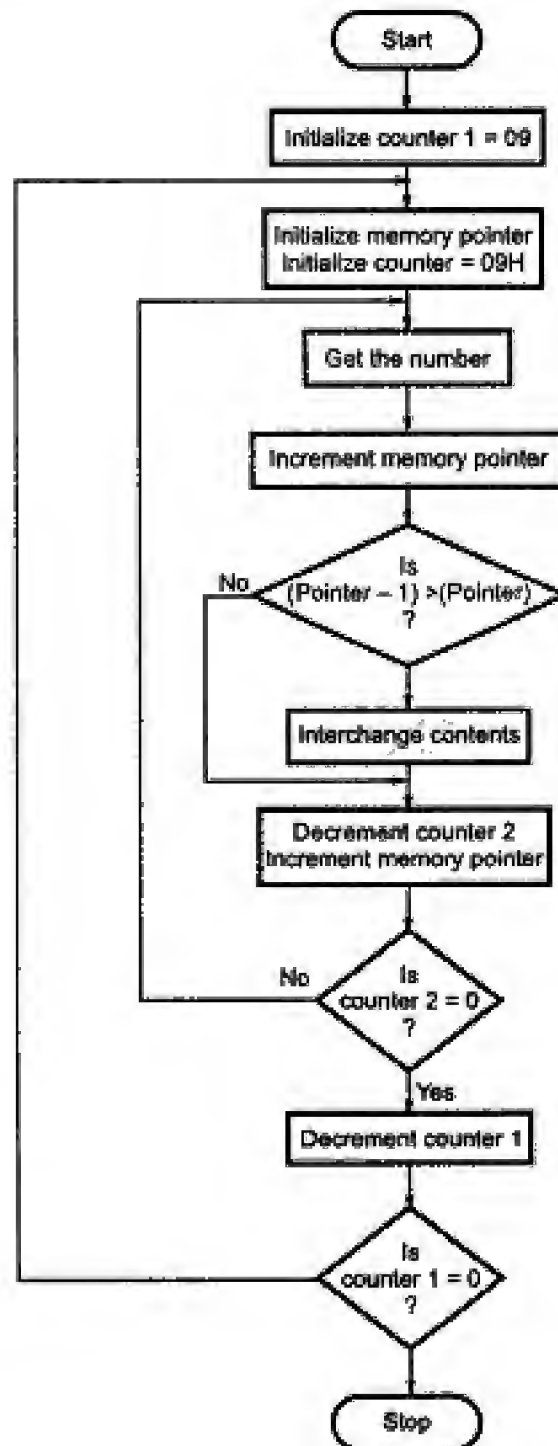
      MVI B, 09      ; Initialize counter 1
START: LXI H, 2200H  ; Initialize memory pointer
      MVI C, 09H     ; Initialize counter 2
BACK : MOV A, M      ; Get the number
      INX H          ; Increment memory pointer
      CMP M          ; Compare number with next number
      JC SKIP        ; If less, don't interchange
      JZ SKIP        ; If equal, don't interchange
      MOV D, M
      MOV M, A
      DCX H
  
```

```

MOV M, D
INX H           ; Interchange two numbers
SKIP : DCR C    ; Decrement counter 2
JNZ BACK       ; If not zero, repeat
DCR B          ; Decrement counter 1
JNZ START      ; If not zero, repeat
HLT            ; Terminate program execution

```

Flowchart



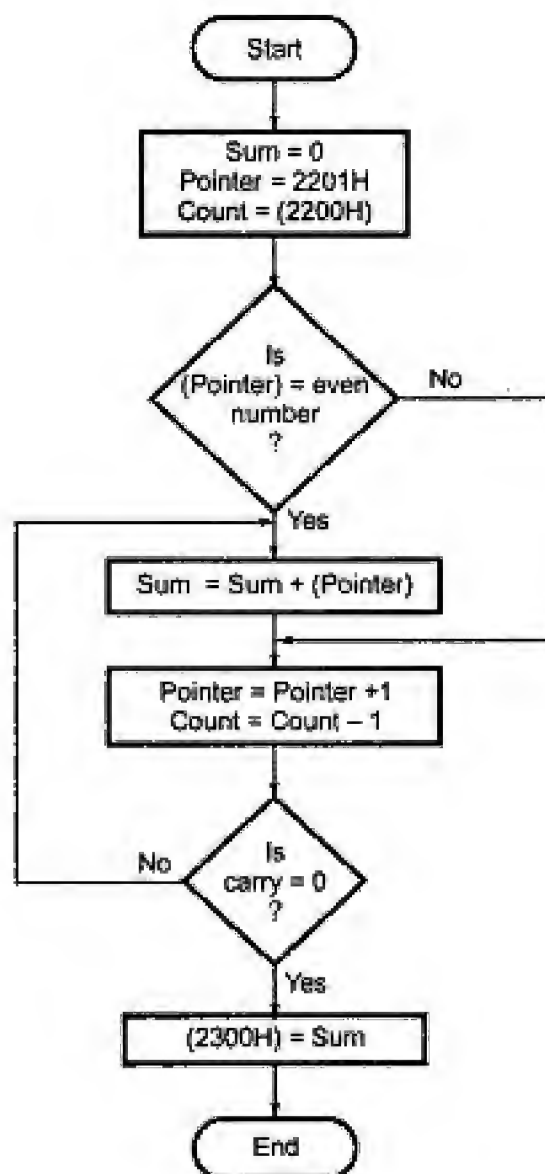
Lab Experiment 30 : Calculate the sum of series of even numbers.

Statement : Calculate the sum of series of even numbers from the list of numbers. The length of the list is in memory location 2200H and the series itself begins from memory location 2201H. Assume the sum to be 8 bit number so you can ignore carries and store the sum at memory location 2210H.

Sample problem :

| | | | |
|--------|---|---------|-------|
| 2200H | = | 4H | |
| 2201H | = | 20H | |
| 2202H | = | 15H | |
| 2203H | = | 13H | |
| 2204H | = | 22H | |
| Result | = | 20 + 22 | = 42H |
| ∴ | | | |
| 2210H | = | 42H | |

Flowchart :



Source program :

```

        LDA 2200H
        MOV C, A      ; Initialize counter
        MVI B, 00H    ; sum = 0
        LXI H, 2201H  ; Initialize pointer
BACK :   MOV A, M      ; Get the number
        ANI 01H       ; Mask Bit1 to Bit7
        JNZ SKIP      ; Don't add if number is ODD
        MOV A, B       ; Get the sum
        ADD M         ; SUM = SUM + data
        MOV B, A      ; Store result in B register
SKIP :   INX H         ; increment pointer
        DCR C         ; Decrement counter
        JNZ BACK      ; if counter 0 repeat
        STA 2210H     ; store sum
        HLT           ; Terminate program execution

```

Lab Experiment 31 : Calculate the sum of series of odd numbers.

Statement : Calculate the sum of series of odd numbers from the list of numbers. The length of the list is in memory location 2200H and the series itself begins from memory location 2201H. Assume the sum to be 16-bit. Store the sum at memory locations 2300H and 2301H.

Sample program :

```

2200H    = 4H
2201H    = 9AH
2202H    = 52H
2203H    = 89H
2204H    = 3FH
Result   = 89H + 3FH = C8H
∴ 2300H  = 61H Lower byte
    2301H  = 01H Higher byte

```

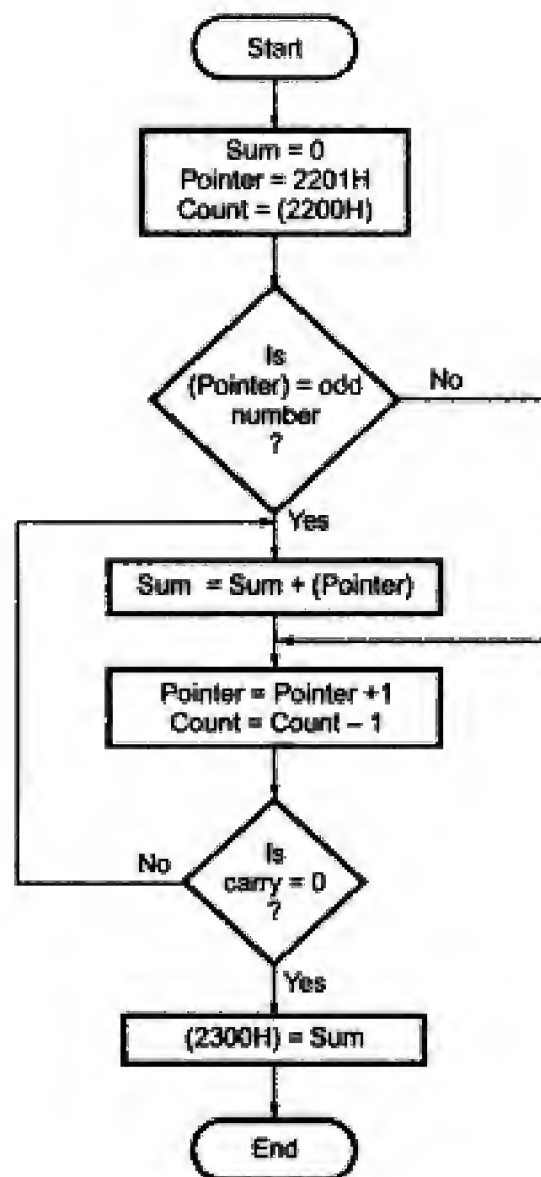
Source program

```

        LDA 2200H
        MOV C, A      ; Initialize counter
        LXI H, 2201H  ; Initialize pointer
        MVI E, 00     ; Sumlow = 0
        MOV D, E      ; Sumhigh = 0
BACK :   MOV A, M      ; Get the number
        ANI 01H       ; Mask Bit1 to Bit7
        JZ SKIP       ; Don't add if number is even
        MOV A, E       ; Get the lower byte of sum
        ADD M         ; Sum = sum + data
        MOV E, A      ; Store result in E register
        JNC SKIP
        INR D          ; Add carry to MSB of SUM
SKIP :   INX H         ; Increment pointer
        DCR C         ; Decrement counter

```

```
JNZ BACK          ; Check if counter ≠ 0 repeat
MOV A, E
STA 2300H          ; Store lower byte
MOV A, D
STA 2301H          ; Store higher byte
HLT                ; Terminate program execution
```

Flowchart

Lab Experiment 32 : Find the square of given number.

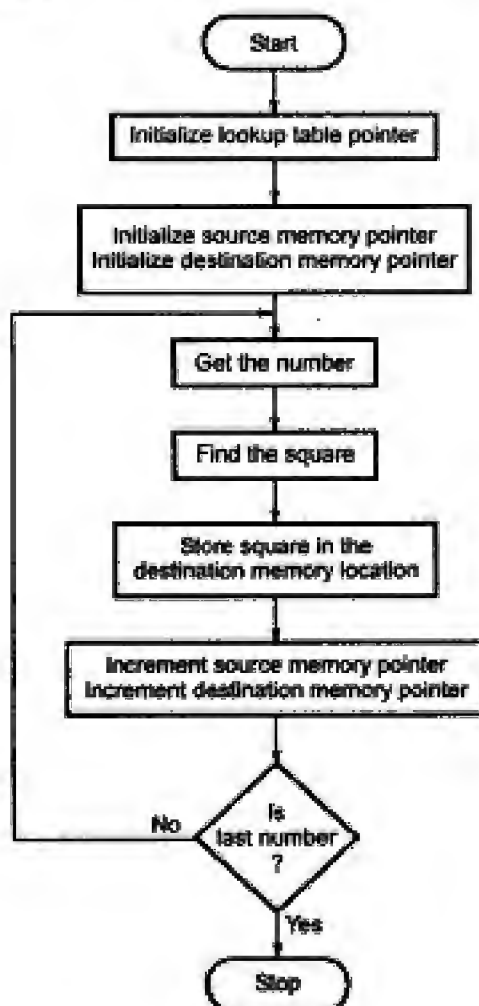
Statement : Find the square of the given numbers from memory location 6100H and store the result from memory location 7000H.

Source Program :

```

        LXI H,6200H    ; Initialize lookup table pointer
        LXI D,6100H    ; Initialize source memory pointer
        LXI B,7000H    ; Initialize destination memory pointer
BACK :  LDAX D          ; Get the number
        MOV L, A        ; A point to the square
        MOV A, M        ; Get the square
        STAX B          ; Store the result at destination
                        ; memory location
        INX D           ; Increment source memory pointer
        INX B           ; Increment destination memory pointer
        MOV A, C        ;
        CPI 05H         ; Check for last number
        JNZ BACK        ; If not repeat
        HLT            ; End of program

```

Flowchart :**Lookup Table**

| Digit | Square |
|-------|--------|
| 0 | 0H |
| 1 | 1H |
| 2 | 4H |
| 3 | 9H |
| 4 | 10H |
| 5 | 19H |
| 6 | 24H |
| 7 | 31H |
| 8 | 40H |
| 9 | 51H |

Lab Experiment 33 : Search a byte in a given number.

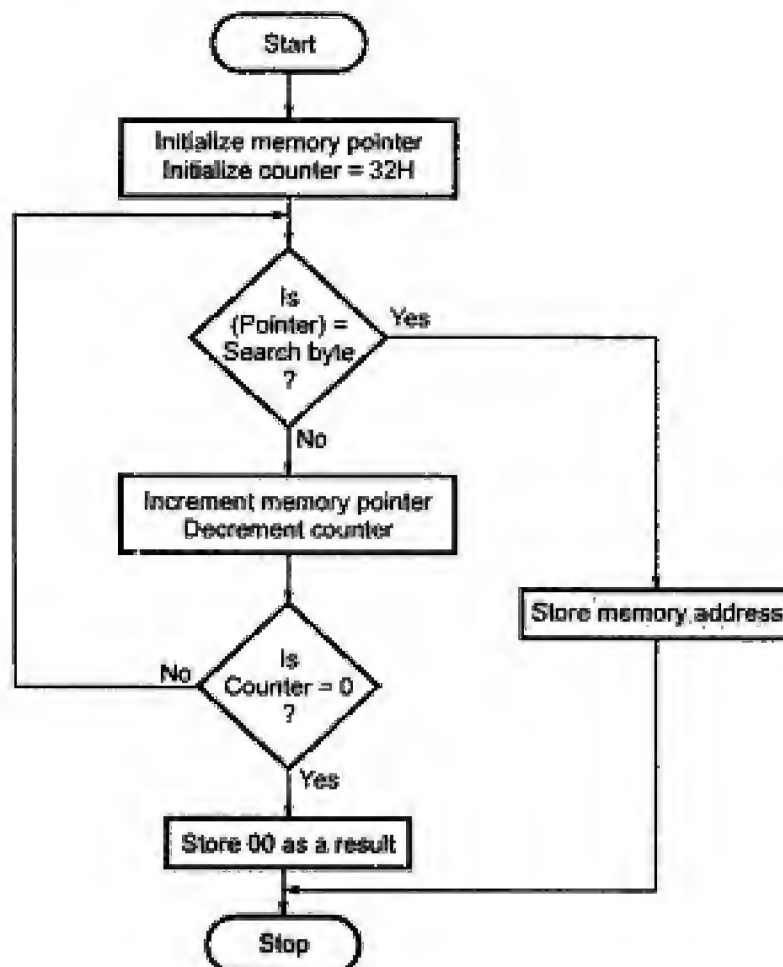
Statement : Search the given byte in the list of 50 numbers stored in the consecutive memory locations and store the address of memory location in the memory locations 2200H and 2201H. Assume byte is in the C register and starting address of the list is 2000H. If byte is not found store 00 at 2200H and 2201H.

Source program :

```

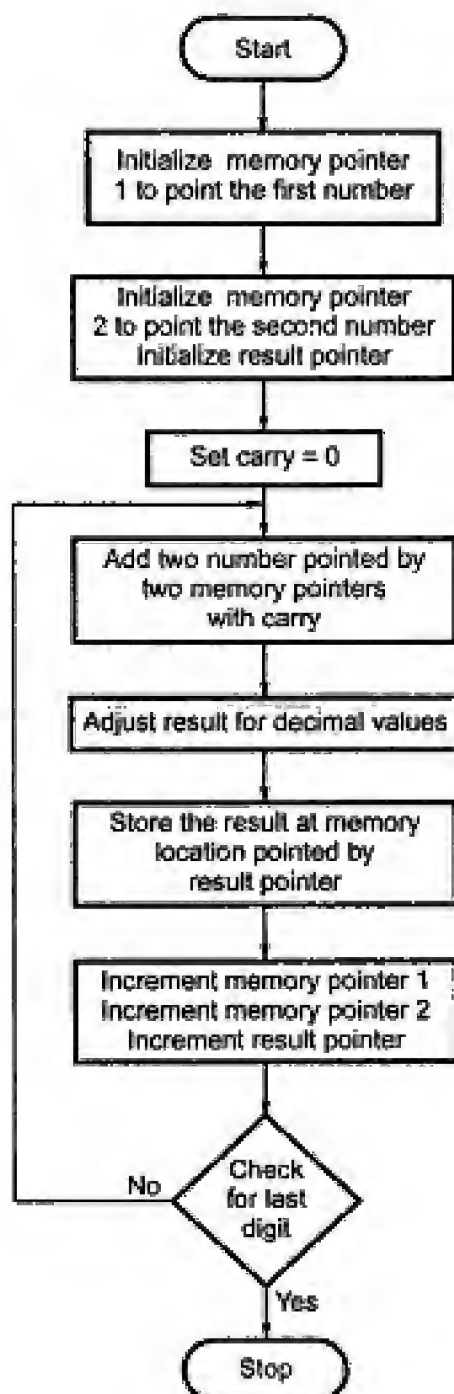
        LXI H, 2000H ; Initialize memory pointer
        MVI B, 52H   ; Initialize counter
BACK :  MOV A, M      ; Get the number
        CMP C        ; Compare with the given byte
        JZ LAST      ; Go last if match occurs
        INX H        ; Increment memory pointer
        DCR B        ; Decrement counter
        JNZ B        ; If not zero, repeat
        LXI H, 0000H
        SHLD 2200H
        JMP END      ; Store 00 at 2200H and 2201H
LAST :  SHLD 2200H    ; Store memory address
        END :        HLT ; Stop

```

Flowchart :

Lab Experiment 34 : Add two decimal numbers.

Statement : Two decimal numbers six digits each, are stored in BCD packed form. Each number occupies a sequence of byte in the memory. The starting address of first number is 6000H and second number is 6100H. Write an assembly language program that adds these two numbers and stores the sum in the same format starting from memory location 6200H.

Flowchart

Source program :

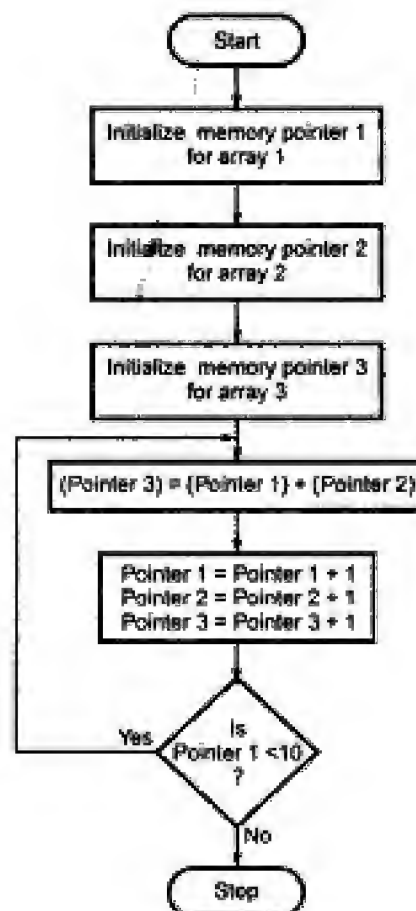
```

        LXI H,6000H    ; Initialize pointer1 to first number
        LXI D,6100H    ; Initialize pointer2 to second number
        LXI B,6200H    ; Initialize pointer3 to result
        STC
        CMC            ; Carry = 0
BACK :   LDAX D         ; Get the digit
        ADD M          ; Add two digits
        DAA            ; Adjust for decimal
        STAX B         ; Store the result
        INX H          ; Increment pointer1
        INX D          ; Increment pointer2
        INX B          ; Increment result pointer
        MOV A, L
        CPI 06H        ; Check for last digit
        JNZ BACK       ; If not last digit repeat
        HLT            ; Terminate program execution

```

Lab Experiment 35 : Add each element of array with the elements of another array.

Statement : Add 2 arrays having ten 8-bit numbers each and generate a third array of result. It is necessary to add the first element of array1 with the first element of array-2 and so on. The starting addresses of array1, array2 and array3 are 2200H, 2300H and 2400H, respectively.

Flowchart

Source program :

```

        LXI H, 2200H ; Initialize memory pointer 1
        LXI B, 2300H ; Initialize memory pointer 2
        LXI D, 2400H ; Initialize result pointer
BACK :   LDAX B       ; Get the number from array 2
        ADD M        ; Add it with number in array 1
        STAX D       ; Store the addition in array 3
        INX H        ; Increment pointer1
        INX B        ; Increment pointer2
        INX D        ; Increment result pointer
        MOV A, L     ;
        CPI 0AH      ; Check pointer1 for last number
        JNZ BACK     ; If not, repeat
        HLT          ; Stop

```

Lab Experiment 36 : Separate even numbers from given numbers.

Statement : Write an assembly language program to separate even numbers from the given list of 50 numbers and store them in the another list starting from 2300H. Assume starting address of 50 number list is 2200H.

Flowchart (See on next page)

Source program :

```

        LXI H, 2200H ; Initialize memory pointer1
        LXI D, 2300H ; Initialize memory pointer2
        MVI C, 32H   ; Initialize counter
BACK:   MOV A,M       ; Get the number
        ANI 01H      ; Check for even number
        JNZ SKIP     ; If ODD, don't store
        MOV A,M       ; Get the number
        STAX D       ; Store the number in result list
        INX D        ; Increment pointer 2
SKIP:   INX H         ; Increment pointer1
        DCR C        ; Decrement counter
        JNZ BACK     ; If not zero, repeat
        HLT          ; Stop

```

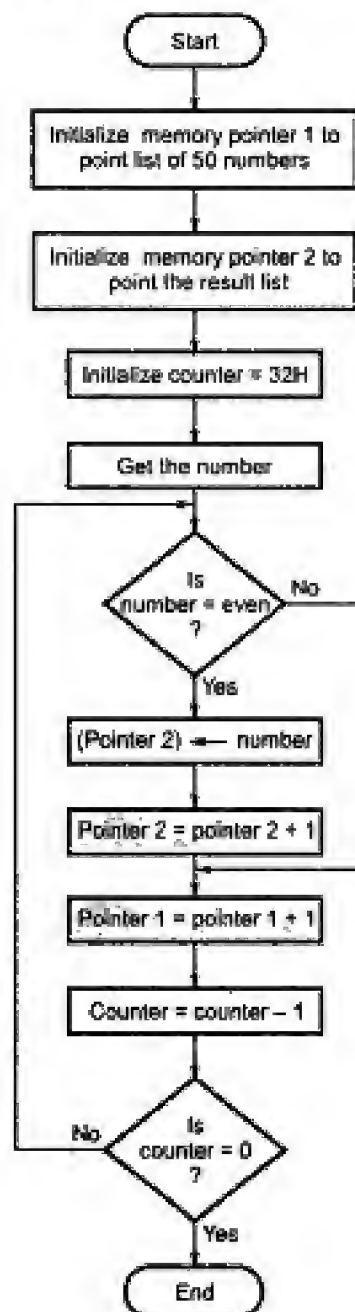
Lab Experiment 37 : Transfer contents to overlapping memory blocks.

Statement : Write assembly language program to with proper comments for the following : A block of data consisting of 256 bytes is stored in memory starting at 3000H. This block is to be shifted (relocated) in memory from 3050H onwards. Do not shift the block or part of the block anywhere else in the memory.

Source program :

Two blocks (3000 – 30FF and 3050 – 314F) are overlapping. Therefore it is necessary to transfer last byte first and first byte last.

Flowchart (For Lab Experiment 36)



```

MVI C, FFH      ; Initialize counter
LXI H, 30FFH    ; Initialize source memory pointer
LXI D, 314FH    ; Initialize destination memory pointer
BACK:  MOV A, M   ; Get byte from source memory block
       STAX D     ; Store byte in the destination memory
               ; block
       DCX H      ; Decrement source memory pointer
       DCX D      ; Decrement destination memory pointer
       DCR C      ; Decrement counter
       JNZ BACK   ; If counter ≠ 0 repeat
       HLT        ; Stop execution
  
```

Lab Experiment 38 : Inserting string in a given array of characters.

Statement : Write an 8085 assembly language program to insert a string of four characters from the tenth location in the given array of 50 characters.

Solution :

Step 1 : Move bytes from location 10 till the end of array by four bytes downwards.

Step 2 : Insert four bytes at locations 10, 11, 12 and 13.

```

        LXI H, 2131H      ; Initialize pointer at the last location
                          ; of array.
        LXI D, 2135H      ; Initialize another pointer to point the
                          ; last location of array after insertion.
AGAIN :  MOV A, M          ; Get the character
        STAX D             ; Store at the new location
        DCX D             ; Decrement destination pointer
        DCX H             ; Decrement source pointer
        MOV A, L          ; [ check whether desired bytes are
        CPI 08H           ; shifted or not]
        JNZ AGAIN         ; if not repeat the process
        INX H             ; adjust the memory pointer
        LXI D, 2200H      ; Initialize the memory pointer to point
                          ; the string to be inserted
REPE :   LDAX D            ; Get the character
        MOV M, A          ; Store it in the array
        INX D             ; Increment source pointer
        INX H             ; Increment destination pointer
        MOV A, E          ; [ check whether the 4 bytes
        CPI 04            ; are inserted]
        JNZ REPE         ; if not repeat the process
        HLT              ; stop

```

Lab Experiment 39 : Deleting string in a given array of characters.

Statement : Write an 8085 assembly language program to delete a string of 4 characters from the tenth location in the given array of 50 characters.

Solution : Shift bytes from location 14 till the end of array upwards by 4 characters i.e. from location 10 onwards.

```

        LXI H, 210DH      ; Initialize source memory pointer at the 14th
                          ; location of the array.
        LXI D, 2109H      ; Initialize destination memory pointer at the
                          ; 10th location of the array.
        MOV A, M          ; Get the character
        STAX D            ; Store character at new location
        INX D             ; Increment destination pointer
        INX H             ; Increment source pointer
        MOV A, L          ; [ check whether desired

```



```

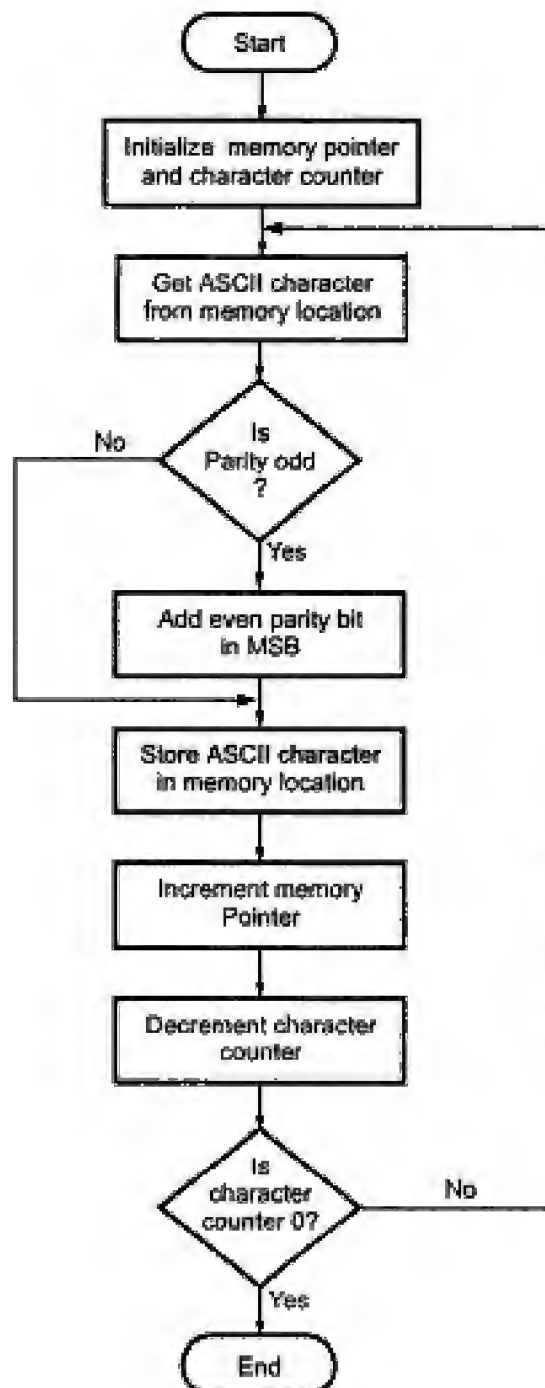
CPI 32H          ; bytes are shifted or not]
JNZ REPE         ; if not repeat the process.
HLT              ; stop

```

Lab Experiment 40 : Add parity bit to 7-bit ASCII characters.

Statement : Add even parity to a string of 7-bit ASCII characters. The length of the string is in memory location 2040H and the string itself begins in memory location 2041H. Place even parity in the most significant bit of each character. Draw a flowchart and write an 8085 assembly language program with comment for each instruction.

Flowchart



Source program :

```

        LXI H, 2040H
        MOV C,M      ; Counter for character
REPEAT: INX H        ; Memory pointer to character
        MOV A,M      ; Character in accumulator
        ORA A        ; ORing with itself to check parity.
        JPO PAREVEN
        ORI 80H      ; If odd parity place even parity in
                    ; D7(80).
PEREVEN: MOV M,A     ; Store converted even parity character.
        DCR C        ; Decrement counter.
        JNZ REPEAT   ; If not zero go for next character.
        HLT          ; Terminate program execution

```

Lab Experiment 41 : Find the number of negative, zero and positive numbers.

Statement : A list of 50 numbers is stored in memory, starting at 6000H. Find number of negative, zero and positive numbers from this list and store these results in memory locations 7000H, 7001H, and 7002H respectively.

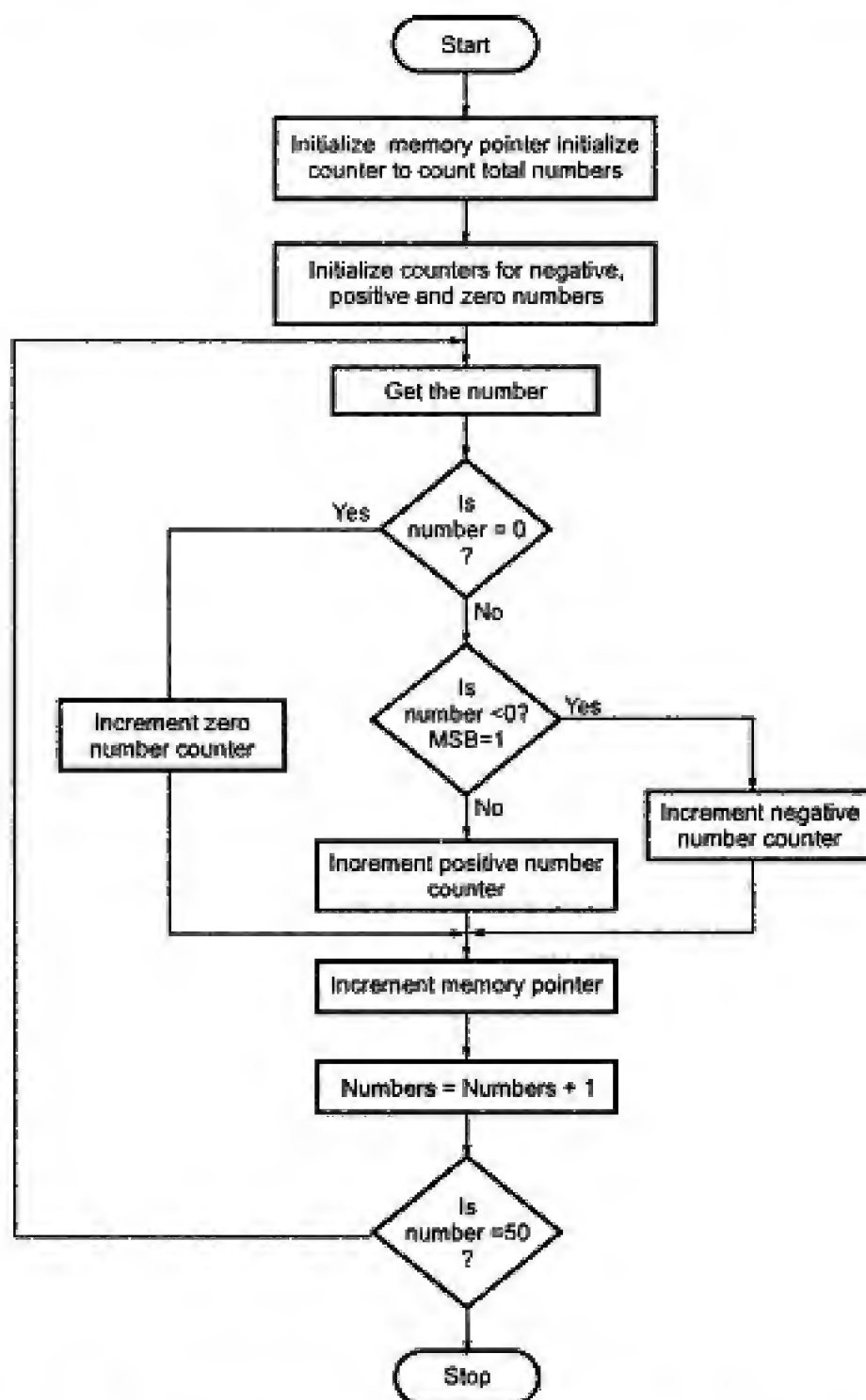
Source program

```

        LXI H, 6000H ; Initialize memory pointer
        MVI C, 00H   ; Initialize number counter
        MVI B, 00H   ; Initialize negative number counter
        MVI E, 00H   ; Initialize zero number counter
BEGIN:  MOV A, M      ; Get the number
        CPI 00H      ; If number = 0
        JZ ZERONUM   ; Goto zeronum
        ANI 80H      ; If MSB of number = 1 i.e. if
        JNZ NEGNUM   ; number is negative goto NEGNUM
        INR D        ; otherwise increment positive number
                    ; counter
        JMP LAST
ZERONUM: INR E        ; Increment zero number counter
        JMP LAST
NEGNUM:  INR B        ; Increment negative number counter
LAST:   INX H        ; Increment memory pointer
        INR C        ; Increment number counter
        MOV A, C
        CPI 32H      ; If number counter = 5010 then
        JNZ BEGIN    ; Store otherwise check next number
        LXI H, 7000  ; Initialize memory pointer.
        MOV M, B     ; Store negative number.
        INX H
        MOV M, E     ; Store zero number.
        INX H
        MOV M, D     ; Store positive number.
        HLT

```

Flowchart

**Lab Experiment 42 : Multiply two eight bit numbers with shift and add method.**

Statement : Multiply the 8-bit unsigned number in memory location 2200H by the 8-bit unsigned number in memory location 2201H. Store the 8 least significant bits of the result in memory location 2300H and the 8 most significant bits in memory location 2301H.

Sample problem

| | | | |
|--------------------|-------------------|-------------------------|--|
| (2200) | = | 1100 (0CH) | |
| (2201) | = | 0101 (05H) | |
| Multiplicand | Multiplier | Result | |
| 1100 (12_{10}) | 0101 (5_{10}) | $12 \times 5 = 60_{10}$ | |

For simplicity, multiplicand and multiplier are taken 4-bit each.

| Steps | Product | Multiplier | | Comments |
|--------|---|------------|---|-------------------------------|
| | B ₇ B ₆ B ₅ B ₄ B ₃ B ₂ B ₁ B ₀ | CY | B ₃ B ₂ B ₁ B ₀ | |
| | 0 0 0 0 0 0 0 0 | 0 | 0 1 0 1 | Initial stage |
| Step 1 | 0 0 0 0 0 0 0 0 | 0 | 1 0 1 0 | Shift left by 1 |
| | 0 0 0 0 0 0 0 0 | 0 | 1 0 1 0 | Don't Add since CY = 0 |
| Step 2 | 0 0 0 0 0 0 0 0 | 1 | 0 1 0 0 | Shift |
| | 0 0 0 0 1 1 0 0 | 1 | 0 1 0 0 | Add multiplicand Since CY = 1 |
| Step 3 | 0 0 0 1 1 0 0 0 | 0 | 1 0 0 0 | Shift left by 1 |
| | 0 0 0 1 1 0 0 0 | 0 | 1 0 0 0 | Don't Add since CY=0 |
| Step 4 | 0 0 1 1 0 0 0 0 | 1 | 0 0 0 0 | Shift left by 1 |
| | 0 0 1 1 1 1 0 0 | 1 | 0 0 0 0 | Add multiplicand Since CY = 1 |

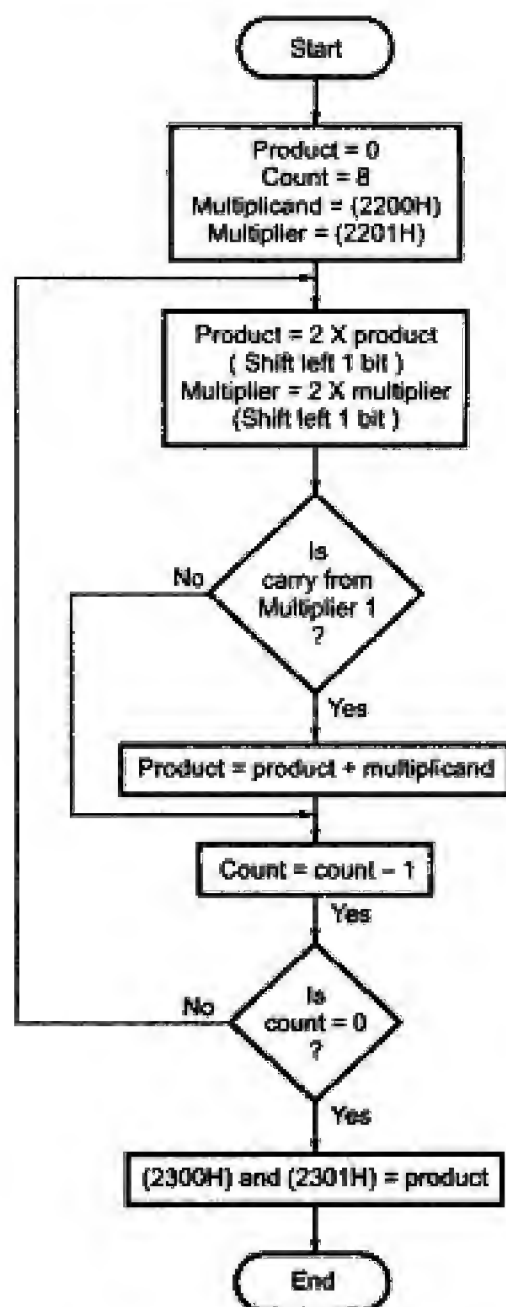
Source program

```

      LXI H, 2200H ; Initialize the memory pointer
      MOV E, M     ; Get multiplicand
      MVI D, 00H   ; Extend to 16-bits
      INX H        ; Increment memory pointer
      MOV A, M     ; Get multiplier
      LXI H, 0000H ; Product = 0
      MVI B, 06H   ; Initialize counter with count 6
MULT:  DAD H        ; Product = product x 2
      RAL
      JNC SKIP     ; Is carry from multiplier 1 ?
      DAD D        ; Yes, Product =Product + Multiplicand
SKIP:  DCR B       ; Is counter = zero
      JNZ MULT     ; no, repeat
      SHLD 2300H   ; Store the result
      HLT         ; End of program

```

Flowchart :



Lab Experiment 43 : Divide 16-bit number with 8-bit number using shifting technique.

Statement : Divide the 16-bit unsigned number in memory locations 2200H and 2201H (most significant bits in 2201H) by the 8-bit unsigned number in memory location 2300H store the quotient in memory location 2400H and remainder in 2401H.

Assumption : The most significant bits of both the divisor and dividend are zero.

Sample problem :

For simplicity, dividend and divisor are taken 8-bit and 4-bit respectively

Dividend = 0110 0001 (61H) Divisor = 0111 (07H)

| Steps | Dividend | Quotient | Comment |
|--------|---|---|--|
| | B ₇ B ₆ B ₅ B ₄ B ₃ B ₂ B ₁ B ₀ | B ₃ B ₂ B ₁ B ₀ | |
| | 0 1 1 0 0 0 0 1 | 0 0 0 0 | Initial stage |
| Step 1 | 1 1 0 0 0 0 1 0 | 0 0 0 0 | Shift left |
| | 0 1 0 1 0 0 1 0 | 0 0 0 1 | MSB of dividend = MSB dividend – divisor and Quotient = Quotient + 1 since MSB of dividend > divisor |
| Step 2 | 1 0 1 0 0 1 0 0 | 0 0 1 0 | Shift left |
| | 0 0 1 1 0 1 0 0 | 0 0 1 1 | MSB of dividend = MSB dividend – divisor and Quotient = Quotient + 1 since MSB of dividend > divisor |
| Step 3 | 0 1 1 0 1 0 0 0 | 0 1 1 0 | Shift left |
| | 0 1 1 0 1 0 0 0 | 0 1 1 0 | No change since MSB of dividend < divisor |
| Step 4 | 1 1 0 1 0 0 0 0 | 1 1 0 0 | Shift left |
| | 0 1 1 0 0 0 0 0 | 1 1 0 1 | MSB of dividend = MSB dividend – divisor and Quotient = Quotient + 1 since MSB of dividend > divisor |

Source program

```

        MVI E, 00      ; Quotient = 0
        LHLD 2200H     ; Get dividend
        LDA 2300        ; Get divisor
        MOV B, A        ; Store divisor
        MVI C, 08      ; Count = 8
NEXT:    DAD H          ; Dividend = Dividend × 2
        MOV A, E
        RLC
        MOV E, A        ; Quotient = × 2
        MOV A, H
        SUB B           ; Is most significant byte of Dividend
                        ; > divisor
        JC SKIP         ; No, go to Next step
        MOV H, A        ; Yes, subtract divisor
        INR E           ; and Quotient = Quotient + 1
SKIP:    DCR C          ; Count = Count - 1
        JNZ NEXT       ; Is count = 0 repeat
        MOV A, E
        STA 2401H      ; Store Quotient

```

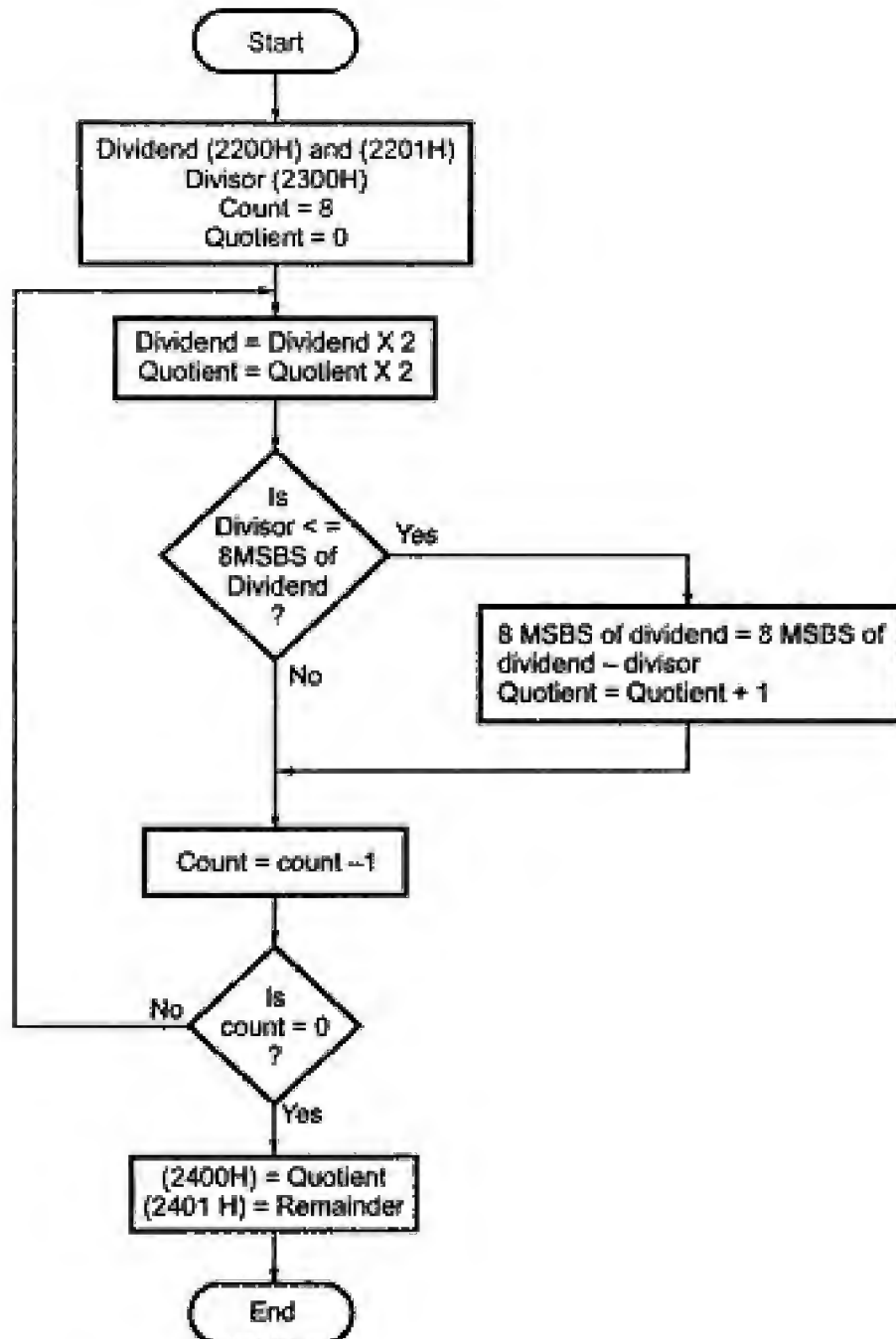


```

MOV A, H
STA 2401H      ; Store remainder
HLT           ; End of program.

```

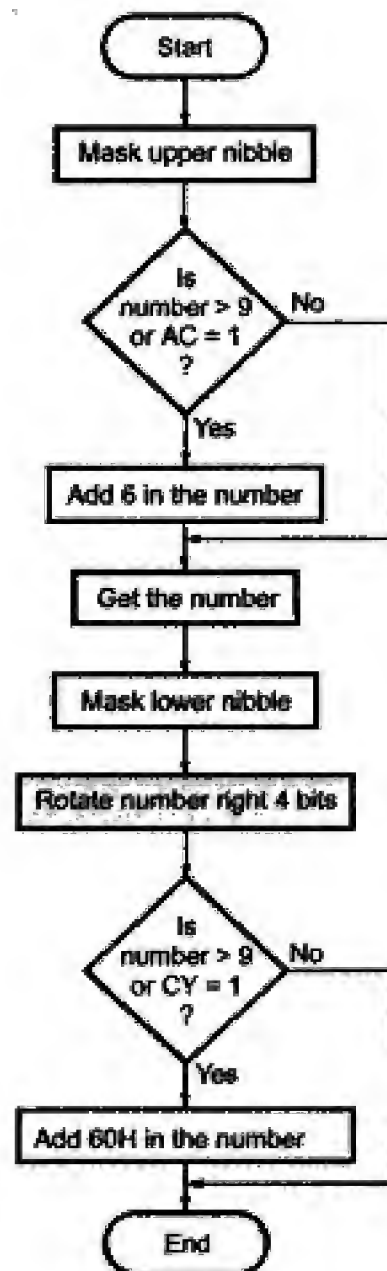
Flowchart :



Lab Experiment 44 : Simulate DAA instruction.

Statement : Assume the DAA instruction is absent. Write a subroutine which will perform the same task as DAA instruction

Flowchart :



Sample problem :

Let us see the execution of DAA instruction.

1. If the value of the low order four bits ($D_3 - D_0$) in the accumulator is greater than 9 or if auxiliary carry flag is set, the instruction adds 6 (06) to the low-order four bits.
2. If the value of the high-order four bits ($D_7 - D_4$) in the accumulator is greater than 9 or if carry flag is set, the instruction adds 6(06) to the high-order four bits.

Note : To check auxiliary carry flag it is necessary to get the flag register contents in one of the registers and then we can check the auxiliary carry flag by checking bit 4 of that register. To get the flag register contents in any general purpose register we require stack operation and therefore stack pointer is initialized at the beginning of the source program.

Source program :

```

        LXI SP, 27FFH      ; Initialize stack pointer
        MOV E,A            ; Store the contents of accumulator
        ANI 0FH            ; Mask upper nibble
        CPI 0A H           ; Check if number is greater than 9
        JC SKIP            ; if no go to skip
        MOV A, E           ; Get the number
        ADI 06H            ; Add 6 in the number
        JMP SECOND         ; Go for second check
SKIP:   PUSH PSW           ; Store accumulator and flag contents
        ; in stack
        POP B              ; Get the contents of accumulator in B
        ; register and
        ; flag register contents in C register
        MOV A, C           ; Get flag register contents in
        ; accumulator
        ANI 10H            ; Check for bit 4
        JZ SECOND         ; if zero, go for second check
        MOV A, E           ; Get the number
        ADI 06             ; Add 6 in the number
SECOND: MOV E, A           ; Store the contents of accumulator
        ANI 0FH            ; Mask lower nibble
        RRC
        RRC
        RRC
        RRC                ; Rotate number 4 bit right
        CPI 0A H           ; Check if number is greater than 9
        JC SKIP1           ; if no go to skip 1
        MOV A, E           ; Get the number
        ADI 60 H           ; Add 60 H in the number
        JMP LAST           ; Go to last
SKIP1:  JNC LAST           ; if carry flag = 0 go to last
        MOV A, E           ; Get the number
        ADI 60 H           ; Add 60 H in the number
LAST:   HLT

```

Lab Experiment 45 : Program to test RAM.

Statement : Write an assembly language program with proper comments to perform following operations :

To test RAM by writing '1' and reading it back and later writing '0' (zero) and reading it back. RAM addresses to be checked are 40FFH to 40FFH. In case of any error, it is indicated by writing 01H at port 10H.

Source program :

```

        LXI H, 4000H ; Initialize memory pointer
BACK :  MVI M, FFH   ; Writing '1' into RAM
        MOV A, M     ; Reading data from RAM
        CPI FFH      ; Check for ERROR
        JNZ ERROR    ; If yes go to ERROR
        INX H        ; Increment memory pointer
        MOV A, H
        CPI 50H      ; Check for last check
        JNZ BACK     ; If not last repeat
        LXI H, 4000H ; Initialize memory pointer
BACK1 : MVI M, 00H   ; Writing '0' into RAM
        MOV A, M     ; Reading data from RAM
        CPI 00H      ; Check for ERROR
        INX H        ; Increment memory pointer
        MOV A, H
        CPI 50H      ; Check for last check
        JNZ BACK1    ; If not last, repeat
        HLT          ; Stop execution

```

Lab Experiment 46 : Write an assembly language program to generate fibonacci number.

```

        MVI D, COUNT ; Initialise counter
        MVI B, 00     ; Initialize variable to store previous
                        ; number
        MVI C, 01     ; Initialize variable to store current
                        ; number
BACK :  MOV A, B       ; [Add two
        ADD C ;       numbers ]
        MOV B, C      ; Current number is now previous number
        MOV C, A      ; Save result as a new current number
        DCR D         ; Decrement count
        JNZ BACK      ; if count ≠ 0 go to BACK
        HLT           ; Stop.

```

4.2 Timers

In the real time applications, such as traffic light control, digital clock, process control, serial communication, it is important to keep a track with time. For example in traffic light control application, it is necessary to give time delays between two transitions. These time delays are in few seconds and can be generated with the help of executing group of instructions number of times. This software timers are also called time delays or software delays. Let us see how to implement these time delays or software delays.

As you know microprocessor system consists of two basic components, Hardware and software. The software component controls and operates the hardware to get the desired output with the help of instructions. To execute these instructions, microprocessor takes fix time as per the instruction, since it is driven by constant frequency clock. This makes it

possible to introduce delay for specific time between two events. In the following section we will see different delay implementation techniques.

4.2.1 Timer Delay using NOP Instruction

NOP instruction does nothing but takes 4T states of processor time to execute. So by executing NOP instruction in between two instructions we can get delay of 4 T-state

$$1 \text{ T state} = \frac{1}{\text{Operating frequency of 8085}}$$

4.2.2 Timer Delay using Counters

Counting can create time delays. Since the execution times of the instructions used in a counting routine are known, the initial value of the counter, required to get specific time delay can be determined.

Using 8 bit counter :

| | | Number of T-states |
|--------|--------------------------------------|--------------------|
| | MVI C, count ; Load count | 7 T-states |
| BACK : | DCR C ; Decrement count | 4 T-states |
| | JNZ BACK ; If count \neq 0, repeat | 10/7 T-states |

In this program, the instructions DCR C and JNZ BACK execute number of times equal to count stored in the C register. The time taken by this program for execution can be calculated with the help of T-states. The column to the right of the comments indicates the number of T-states in the instruction cycle of each instruction. Two values are specified for the number of T-states for the JNZ instruction. The smaller value is applied when the condition is not met, and the larger value applied when it is met. The first instruction MVI C, count executed only once and it requires 7 T-states. There are count -1 passes through the loop where the condition is met and control is transferred back to the first instruction in the loop (DCR C). The number of T-states that elapse while C is not zero are (count -1) \times (4+10). On the last pass through the loop, the condition is not met and the loop is terminated. The number of T states that elapse in this pass are 4 + 7.

\therefore Total T-states required to execute the given program

$$\begin{aligned}
 &= \underset{\text{MVI C}}{7} + \underset{\text{Loops}}{(\text{count} - 1) \times (4 + 10)} + \underset{\text{Last loop}}{(4 + 7)} \\
 \text{For count} &= 5 \\
 \text{Number of T-state} &= 7 + (5 - 1) \times (14) + (11) \\
 &= 7 + 56 + 11 \\
 &= 74
 \end{aligned}$$

Assuming operating frequency of 8085A is 2 MHz,

$$\begin{aligned}
 \text{Time required for 1 T-state} &= \frac{1}{2 \text{ MHz}} \\
 &= 0.5 \mu\text{sec}
 \end{aligned}$$

Total time required to execute the given program $= 74 \times 0.5 \mu\text{sec}$
 $= 37 \mu\text{sec}$.

Maximum delay possible with 8 bit count

The maximum count that can be loaded in the 8 bit register is FFH (255) so the maximum delay possible with 8 bit count, assuming operating frequency 2 MHz

$$= (7 + (255 - 1) \times (14) + (11)) \times 0.5 \mu\text{sec}$$

$$= 1787 \mu\text{sec}.$$

With these calculations, it can be noticed that delay with 8 bit count suitable for small delays, and not for large delays.

Using 16 bit counter :

| | | Number of T-states |
|--------|---------------------------------------|--------------------|
| | LXI B, count ; load 16 bit count | 10 T-states |
| BACK : | DCX B ; Decrement count | 6 T-states |
| | MOV A, C ; | 4 T-states |
| | ORA B ; logically OR B and C | 4 T-states |
| | JNZ BACK ; If result is not 0, repeat | 10 T-states |

In this program, the instructions DCX B, MOV A,C, ORA B and JNZ BACK execute number of times equal to count stored in BC register pair. The instruction LXI B, count is executed only once. It requires 10T-states. The number of T-states required for one loop $= 6 + 4 + 4 + 10 = 24$ T-states. The number of T-states required for last loop $= 6 + 4 + 4 + 7 = 21$ T-states. So total T-states required for execution of given program are

$$= 10 \quad + \quad (\text{count}-1) \quad \times \quad 24 + 21$$

$$\text{LXI B} \quad \quad \text{Loops} \quad \quad \text{Last loop}$$

$$\text{for count} = 03\text{FFH} (1023_{10})$$

$$\text{Number of T-states} = 10 + (1022) \times 24 + 21$$

$$= 24559$$

Assuming operating frequency of 8085 A as 2 MHz, the time required for, T state $= 0.5 \mu\text{sec}$.

$$\therefore \text{Total time required to execute the given program}$$

$$= 24559 \times 0.5 \mu\text{sec}$$

$$= 12279.5 \mu\text{sec}$$

$$= 12.2795 \text{ msec}$$

Maximum delay possible with 16 bit count

The maximum count that can be loaded in the 16 bit register pair is FFFFH (65535 H). So the maximum delay possible with 8 bit count, assuming operating frequency 2 MHz.

$$= 10(10 + (65535 - 1) \times (24) + (21)) \times 0.5 \mu\text{sec}$$

$$= 0.786425 \text{ sec}$$

If the application requires the delays more than this, then the nested loop technique is used to implement the delays.

4.2.3 Timer Delay using Nested Loops

In this, there are more than one loops. The innermost loop is same as explained above. The outer loop sets the multiplying count to the delays provided by the innermost loop.

| | Number of T states |
|--|--------------------|
| MVI B, Multiplier count ; Initialize multiplier | 7 T-states |
| START: MVI C, Delay count ; initialize delay count | 7 T-states |
| BACK: DCR C ; Decrement delay count | 4 T-states |
| JNZ BACK ; if not 0, repeat | 10/7 T-states |
| DCR B ; decrement multiplier count | 4 T-states |
| JNZ START ; If not 0, repeat | 10/7 T-states |

T-states required for execution of inner loop

$$T_{inner} = 7 + (\text{Delay count} - 1) \times 14 + 11$$

T-states required for execution of the given program

$$= (\text{Multiplier count} - 1) \times (T_{inner} + 14) + 11$$

For delay count = 65H (101) and multiplier count

$$= 51H (81)$$

$$T_{inner} = 7 + (101 - 1) \times 14 + 11$$

$$= 1418$$

Total time required to execute the given program is

$$\begin{aligned} (\text{Operating frequency is 2 MHz}) &= [(81 - 1) \times (1418 + 14) + 11] \times 0.5 \mu\text{sec.} \\ &= 57.2855 \text{ msec.} \end{aligned}$$

Lab Experiment 47 : Generate a delay of 0.4 seconds.

Statement : Write a program to generate a delay of 0.4 sec if the crystal frequency is 5 MHz.

Sol. : In 8085, the operating frequency is half of the crystal frequency,

$$\therefore \text{Operating frequency} = 5/2 = 2.5 \text{ MHz}$$

$$\therefore \text{Time for one T-state} = \frac{1}{2.5 \text{ MHz}} = 0.4 \mu\text{sec}$$

$$\begin{aligned} \text{Number of T-states required} &= \frac{\text{Required Time}}{\text{Time for 1 T-state}} = \frac{0.4 \text{ sec.}}{0.4 \mu\text{sec.}} \\ &= 1 \times 10^6 \end{aligned}$$

Delay program :

```

BACK:  LXI B, count      ; 16-bit count
        DCX B           ; Decrement count
        MOV A, C
        ORA B           ; Logically OR B and C
        JNZ BACK        ; If result is not zero repeat

```

$$1 \times 10^6 = 10 + (\text{count} - 1) \times 24 + 21$$

$$\text{count} = \left(\frac{1 \times 10^6 - 31}{24} \right) + 1 = 41666_{10}$$

count = 41666_{10}
 = $A2C2H$

Lab Experiment 48 : Generate and display binary up counter.

Statement : Write a program for displaying binary up counter. Counter should count numbers from 00 to FFH and it should increment after every 0.5 sec.

Assume operating frequency of 8085 equal to 2 MHz. Display routine is available.

Solution :

```

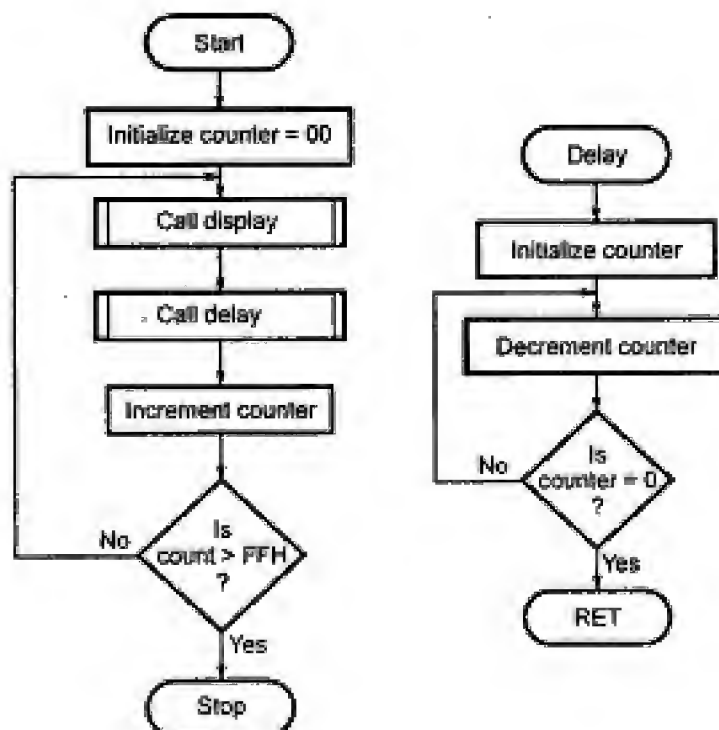
      LXI SP, 27FFH      ; Initialize stack pointer
      MVI C, 00H         ; Initialize counter
BACK:  CALL Display      ; Call display subroutine
      CALL Delay         ; Call delay subroutine
      INR C              ; Increment counter
      MOV A, C           ;
      CPI 00H            ; Check counter is > FFH
      JNZ BACK           ; If not, repeat
      HLT                ; Stop
  
```

Delay subroutine :

```

Delay: LXI D, count      ; Initialize count
BACK:  DCX D             ; Decrement count
      MOV A, E           ;
      ORA D              ; Logically OR D and E
      JNZ BACK           ; If result is not 0 repeat
      RET                ; Return to main program
  
```

Flowchart



Operating frequency = 2 MHz

$$\therefore \text{Time for one T-state} = \frac{1}{2 \text{ MHz}} = 0.5 \mu\text{sec}$$

$$\text{Number of T-states required} = \frac{\text{Required Time}}{\text{Time for 1 T-state}} = \frac{0.5 \text{ sec}}{0.5 \mu\text{sec}} = 1 \times 10^6$$

$$1 \times 10^6 = 10 + (\text{count} - 1) \times 24 + 21$$

$$\text{count} = \left(\frac{1 \times 10^6 - 31}{24} \right) + 1 = 41666_{10}$$

$$\text{count} = 41666_{10} = \text{A2C2H}$$

Lab Experiment 49 : Generate and display BCD up counter with frequency 1 Hz.

Statement : Write a program for displaying BCD up counter. Counter should count numbers from 00 to 99H and it should increment after every 1 sec. Assume operating frequency of 8085 equal to 3 MHz. Display routine is available.

Solution :

```

                LXI SP, 27FFH      ; Initialize stack pointer
                MVI C, 00H         ; Initialize counter
BACK:          CALL Display       ; Call display subroutine
                CALL Delay         ; Call delay subroutine
                MOV A, C           ;
                ADI , 01           ; Increment counter
                DAA                ; Adjust it for decimal
                MOV C, A           ; Store count
                CPI ,00            ; Check count is > 99
                JNZ BACK           ; If not, repeat
                HLT                ; Stop

```

Delay subroutine :

```

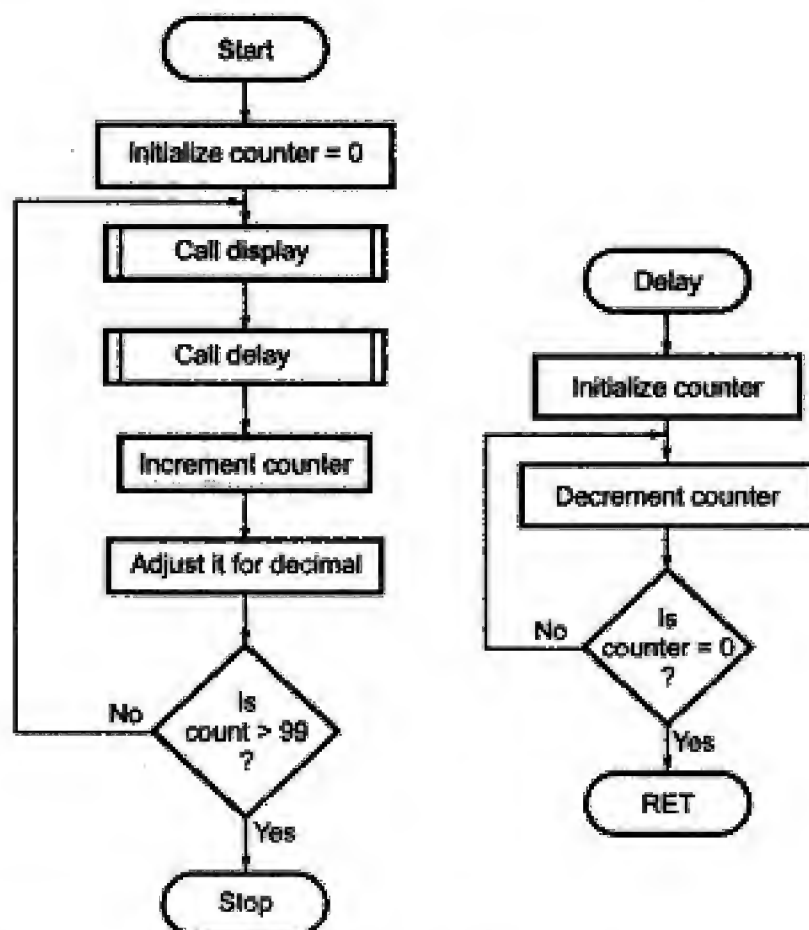
Delay:         MVI B, Multiplier-count ; Initialize multiplier count
BACK1:        LXI D, Initialize Count
BACK:         DCX D                    ; Decrement count
                MOV A, E                ;
                ORA D                    ; Locally OR D and E
                JNZ BACK                ; If result is not 0, repeat
                DCR B                    ; Decrement multiplier count
                JNZ BACK1                ; If not zero, repeat
                RET                      ; Return to main program.

```

Operating frequency : 3 MHz

$$\therefore \text{Time for one T-state} = \frac{1}{3 \text{ MHz}} = 0.333 \mu\text{sec}$$

Flowchart



$$\begin{aligned}
 \therefore \text{Number of T-states required} &= \frac{\text{Required Time}}{\text{Time for 1 - T state}} = 3 \times 10^6 \\
 &= \frac{1 \text{ sec}}{0.333 \mu\text{sec}}
 \end{aligned}$$

Let us take multiplier count = 3.

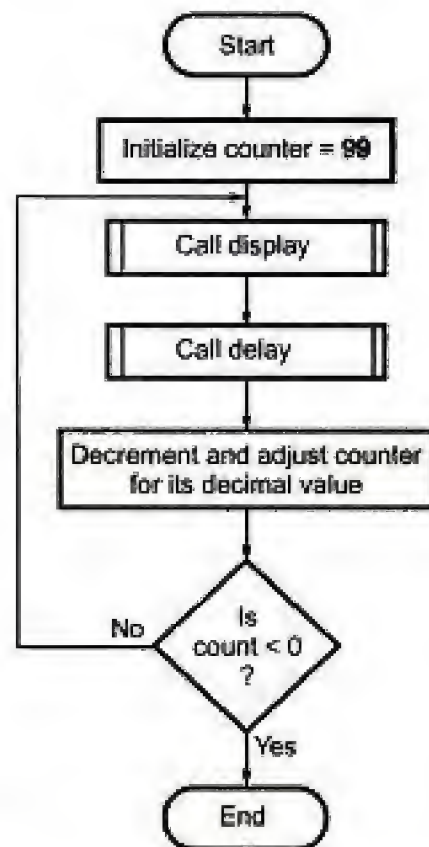
$$\therefore \text{No. of T-states required by inner loop} = \frac{3 \times 10^6}{3} = 1 \times 10^6$$

$$\begin{aligned}
 \therefore 1 \times 10^6 &= 10 + (\text{count} - 1) \times 24 + 21 \\
 &= 4166610 \\
 \text{count} &= 4166610 = \text{A2C2H}
 \end{aligned}$$

Lab Experiment 50 : Generate and display BCD down counter with frequency 1 Hz.

Statement : Write a program for displaying BCD down counter. Counter should count numbers from 99 to 00 and it should decrement after every 1 sec. Assume display and delay routines are available.

Solution : Flowchart :



Source program with logic 1 :

```

                LXI SP, 27FFH      ; Initialize stack pointer
                MVI A, 99H         ; Initialize counter
BACK:           CALL Display       ; Call display subroutine
                CALL Delay         ; Call Delay subroutine
                ADI 99H            ; * (Explained later)
                DAA                ; Adjust for decimal
                CPI 99H            ; Compare with last count
                JNZ BACK           ; If no, repeat
                HLT
  
```

* Addition :

| | | | |
|-------|-------|------|-----|
| | 1001 | 1001 | 99H |
| + | 1001 | 1001 | 99H |
| | 10011 | 0010 | |
| DAA + | 0110 | 0110 | |
| | 1001 | 1000 | 98H |

Program with logic 2 :

```

                LXI SP, 27FFH      ; Initialize stack pointer
                MVI C, 99H         ; Initialize counter = 99
BACK:           Call Display       ; Call display subroutine
                Call Delay         ; Call delay subroutine
  
```

```

        MOV A,C           ; Get the count
        ANI 0FH           ; Check for lower nibble
        JNZ SKIP          ; If it is not 0FH go to skip
        MOV A,C           ; Else get the count
        SBI 06            ; and subtract 6
        MOV C,A           ; Store the count
SKIP:   DCR C              ; Decrement count
        MOV A,C           ; Get the count
        CPI 99H           ; Check it for last count
        JNZ BACK          ; If not, repeat
        HLT               ; Stop

```

Lab Experiment 51 : Generate and display the contents of decimal counter.

Statement : Write assembly language program to with proper comments for the following : To display decimal decrementing counter (99 to 00) at port 05 H with delay of half seconds between each count. Write as well the delay routine giving delay of half seconds. Operating frequency of microprocessor is 3.072 MHz. Neglect delay of the main program.

Source program :

```

        MVI C, 99H        ; Initialize counter
BACK:   MOV A,C            ;
        ANI 0F            ; Mask higher nibble
        CPI 0F
        JNZ SKIP
        MOV A,C
        SUI 06            ; Subtract 6 to adjust decimal count
        MOV D,A
SKIP:   MOV A,C
        OUT 05            ; send count on output port
        CALL Delay        ; Wait for 0.5 seconds
        DCR C             ; decrement count
        MOV A,C
        CPI FF
        JNZ BACK          ; If not zero, repeat
        HLT               ; Stop execution

```

Delay subroutine :

```

Delay:  LXI D, Count
Back::  DCX D              ; 6 T-states
        MOV A, D          ; 4 T-states
        ORA E             ; 4 T-states
        JNZ Back          ; 10 T-states
        RET

```

$$\begin{aligned}
 1 \text{ T-state} &= \frac{1}{\text{Operating frequency}} = \frac{1}{3.072 \times 10^6} \\
 &= 3.2552 \times 10^{-7}
 \end{aligned}$$

$$\text{Number of T-states required} = \frac{0.5 \text{ sec}}{3.2552 \times 10^{-7}} = 1.536 \times 10^6$$

$$1.536 \times 10^6 = 10 + (\text{count} - 1) \times 24 + 21$$

$$\text{Count} = \frac{1.536 \times 10^6 - 31}{24} + 1 = 63999.708$$

$$= 64000$$

$$= \text{FA00H}$$

Lab Experiment 52 : Identify the error and correct the given delay routine.

Statement : The delay routine given below is in infinite loop, identify the error. Correct the program, give the machine cycles and T states of each instruction and also find the maximum delay generated. Assume 1 "T" state = 320 ns.

```

DELAY : LXI H, N
L1      : DCX H
          JNZ L1

```

Solution : 1) The fault in the above program is at instruction JNZ L1. This condition always evaluates to be true hence loops keeps on executing and hence infinite loop.

2) **Reason for infinite looping :** - The instruction DCX H decrease the HL pair count one by one but it does not affect the zero flag. So when count reaches to 0000H in HL pair zero flag is not affected and JNZ L1 evaluates to be true and loop continues. Now HL again decrements below 0000H and HL becomes FFFFH and thus execution continues.

3) The modification in the program is as follows :

| | No. of T states |
|--------------------------------------|-----------------|
| DELAY : LXI H, N ; Load 16 bit count | → 10 T-states |
| L1 : DCX H ; Decrement count | → 6 T-states |
| MOV A, L ; | → 4 T-states |
| ORA H ; logically OR H and L | → 4 T-states |
| JNZ L1 ; If result is not 0 repeat | → 10 T-states |

∴ Total no. of T states required for program execution are

$$= 10 + (\text{count} - 1) \times 24 + 21$$

$$\text{LXI H} + \text{Loops} \quad \text{Last Loop}$$

T-states for maximum delay = ?

Now maximum count that can be loaded in 16 bit register pair is FFFFH (65535H) so

$$[\text{T-states for max. delay} = 10 + (65535 - 1) \times 24 + 21]$$

$$= 1.57247 \times 10^6$$

$$\text{Time for 1 T state} = 320 \text{ ns}$$

$$\therefore \text{Max. delay} = 1.575247 \times 10^6 \times 320 \text{ ns}$$

$$= 0.50407904 \text{ seconds}$$

4.3 Code Conversion

This technique allows programmer to translate a number represented using one coding system to another. For example, when we accept any number from the keyboard it is in ASCII code. But for processing, we have to convert this number in its hex equivalent. The code conversion involves some basic conversions such as

- BCD to binary conversion
- Binary to BCD conversion
- BCD to seven segment code conversion
- Binary to ASCII conversion and
- ASCII to binary conversion

4.3.1 BCD to Binary Conversion

We are more familiar with the decimal number system. But the microprocessor understands the binary/hex number system. To convert BCD number into its binary equivalent we have to use the principle of positional weighting in a given number.

$$\begin{aligned}\text{For example :} \quad 67 &= 6 \times 0AH + 7 \\ &= 3CH + 7 = 43H\end{aligned}$$

To perform above operation it is necessary to separate an 8-bit packed BCD number into two 4-bit unpacked BCD digits : BCD_1 and BCD_2 and then convert each digit into its binary value according to its positions. Finally, add both binary numbers to obtain the binary equivalent of the BCD number. Let us see the program for 2-digit BCD to binary conversion.

Lab Experiment 53 : 2-Digit BCD to binary conversion.

Statement : Convert a 2-digit BCD number stored at memory address 2200H into its binary equivalent number and store the result in a memory location 2300H.

Sample problem

$$\begin{aligned}(2200H) &= 67H \\ (2300H) &= 6 \times 0AH + 7 = 3CH + 7 = 43H\end{aligned}$$

Source program :

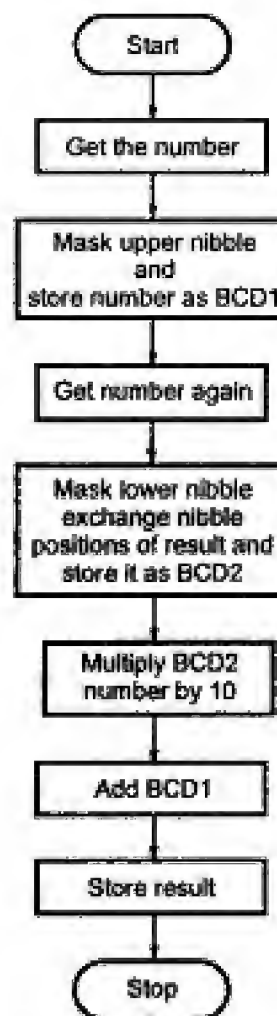
```
LDA 2200H      ; Get the BCD number
MOV B,A        ; Save it
ANI 0FH        ; Mask most significant four bits
MOV C,A        ; Save unpacked BCD1 in C register
MOV A,B        ; Get BCD again
ANI F0H        ; Mask least significant four bits
RRC            ; Convert most significant four
RRC            ; bits into unpacked BCD2
```

```

RRC      ;
RRC      ;
MOV B,A  ; Save unpacked BCD2 in B register
XRA A    ; Clear accumulator (sum = 0)
MVI D, 0AH ; Set D as a multiplier of 10
SUM :    ADD D    ; Add 10 until (B) = 0
        DCR B    ; Decrement BCD2 by one
        JNZ SUM  ; Is multiplication complete ?
        ; if not, go back and add again
        ADD C    ; Add BCD1
        STA 2300H ; Store the result
        HLT      ; Terminate program execution

```

Flowchart



4.3.2 Binary to BCD Conversion

We know that microprocessor processes data in the binary form. But when it is displayed, it is in the BCD form. In this case we need binary to BCD conversion of data. The conversion of binary to BCD is performed by dividing the number by the power of ten.

For example, assume the binary number as

$$0111\ 1011\ (7BH) = 123_{10}$$

To represent the number in BCD requires twelve bits or three BCD digits as shown below

$$123_{10} = 0001\ 0010\ 0011$$

Digit₂ digit₁ digit₀

The conversion can be performed as follows

- Step 1 :** If the number is equal to or greater than 100, divide number by 100 (i.e. subtract 100 repeatedly until the remainder is less than 100). The quotient gives the most significant digit, digit 2 of the BCD number. If number is less than 100 go to step 2.
- Step 2 :** If the number i.e. remainder of first division is equal to or greater than 10 divide number by 10 repeatedly until the remainder is less than 10. The quotient gives the digit 1. If number is less than 10, go to step 3.
- Step 3 :** The remainder from step 2 gives the digit 3.

Let us see the program for binary to BCD conversion.

Lab Experiment 54 : Binary to BCD conversion.

Statement : Write a main program and a conversion subroutine to convert the binary number stored at 6000H into its equivalent BCD number. Store the result from memory location 6100H.

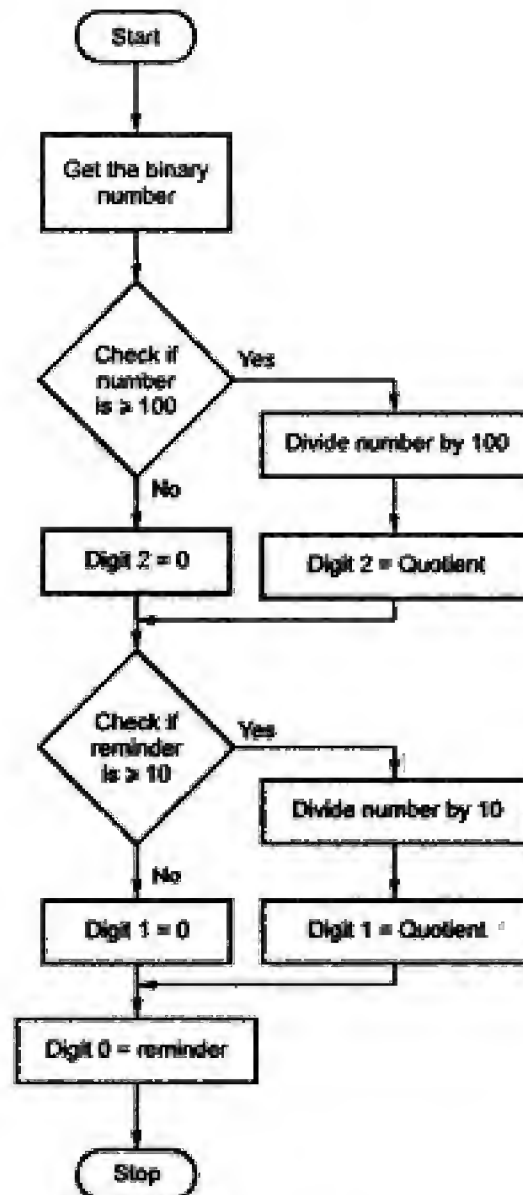
Sample problem : (6000)H = 8AH

1. 8AH \geq 64H (Decimal 100) \therefore Divide by 64H (Decimal 100)
8AH/64H \rightarrow Quotient = 1 Remainder = 26H
26H < 64H (Decimal 100) \therefore goto step 2 and Digit 2 = 1
2. 26H \geq 0AH (Decimal 10) \therefore Divide by 0AH (Decimal 10)
26H/0AH \rightarrow Quotient = 3 Remainder = 08H
08H < 0AH (Decimal 10) \therefore goto step 3 and Digit 1 = 3
3. Digit 0 = 08H

Source program

```
LXI SP, 27FFH    ; Initialize stack pointer
LDA 6000H        ; Get the binary number in accumulator
CALL BIN TO BCD  ; Call subroutine BIN TO BCD
HLT              ; Terminate program execution
```

Flowchart



Subroutine to convert binary number into its equivalent
BCD number

BIN TO BCD :

```

        PUSH B           ; Save BC register pair contents
        PUSH D           ; Save DE register pair contents
        MVI B, 64H       ; Load divisor decimal 100 in B register
        MVI C, 0AH       ; Load divisor decimal 10 in C register
        MVI D, 00H       ; Initialize Digit 1
        MVI E, 00H       ; Initialize Digit 2
STEP1 : CMP B            ; Check if number < Decimal 100
        JC STEP 2        ; if yes go to step 2
        SUB B             ; Subtract decimal 100
        INR E             ; update quotient
        JMP STEP1        ; go to step 1
  
```

```

STEP2 :  CMP C           ; Check if number < Decimal 10
          JC STEP 3      ; if yes go to step 3
          SUB C           ; Subtract decimal 10
          INR D           ; Update quotient
          JMP STEP 2      ; Continue division by 10

STEP3 :  STA 6100H        ; Store Digit 0
          MOV A, D         ; Get Digit 1
          STA 6101H        ; Store Digit 1
          MOV A, E         ; Get Digit 2
          STA 6102H        ; Store Digit 2
          POP D            ; Restore DE register pair contents
          POP B            ; Restore BC register pair contents
          RET              ; Return to main program

```

4.3.3 BCD to Seven Segment Conversion

Many times 7-segment LED display is used to display the results or parameters in the microprocessor system. In such cases we have to convert the result or parameter in 7-segment code. This conversion can be done using look-up technique. In the look-up table the codes of the digits (0-9) to be displayed are stored sequentially in the memory. The conversion program locates the code of a digit based on its BCD digit. Let us see the program for BCD to common cathode 7-segment code conversion.

Lab Experiment 55 : Find the 7-segment codes for given numbers.

Statement : Find the 7-segment codes for given 5 numbers from memory location 6000H and store the result from memory location 7000H.

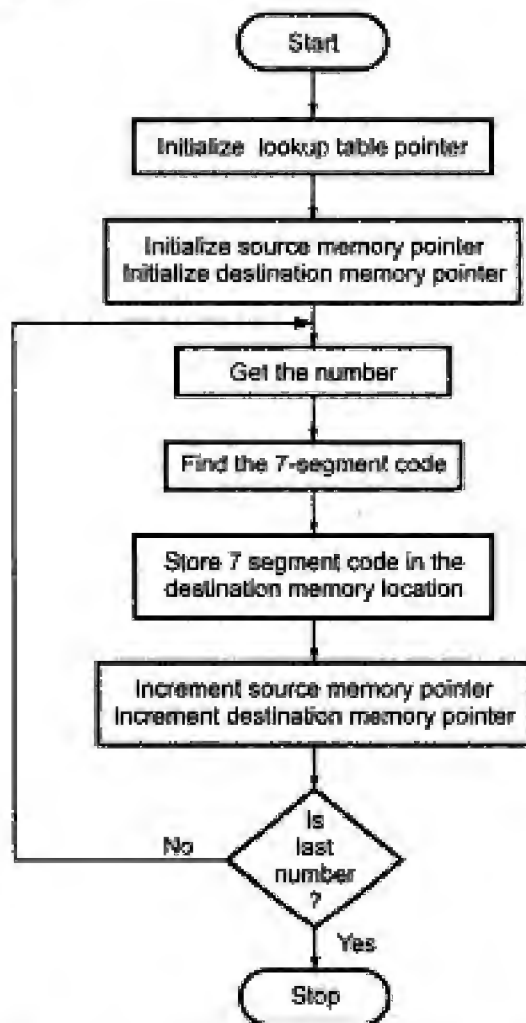
Source program

```

          LXI H, 6200H ; Initialize lookup table pointer
          LXI D, 6000H ; Initialize source memory pointer
          LXI B, 7000H ; Initialize destination memory pointer
BACK:     LDAX D        ; Get the number
          MOV L, A       ; A point to the 7-segment code
          MOV A, M       ; Get the 7-segment code
          STAX B         ; Store the result at destination memory
                      ; location
          INX D          ; Increment source memory pointer
          INX B          ; Increment destination memory pointer
          MOV A, C
          CPI 05H        ; Check for last number
          JNZ BACK       ; If not repeat
          HLT           ; End of program

```


Flowchart :



Lookup Table

| Digit | Code |
|-------|------|
| 0 | 3F |
| 1 | 06 |
| 2 | 5B |
| 3 | 4F |
| 4 | 66 |
| 5 | 6D |
| 6 | 7D |
| 7 | 07 |
| 8 | 7F |
| 9 | 6F |

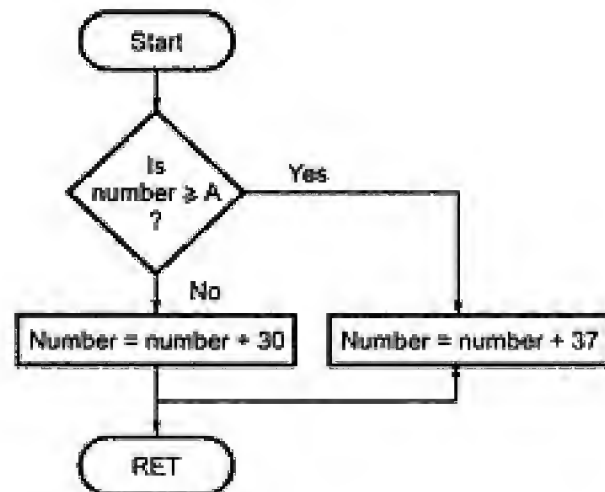
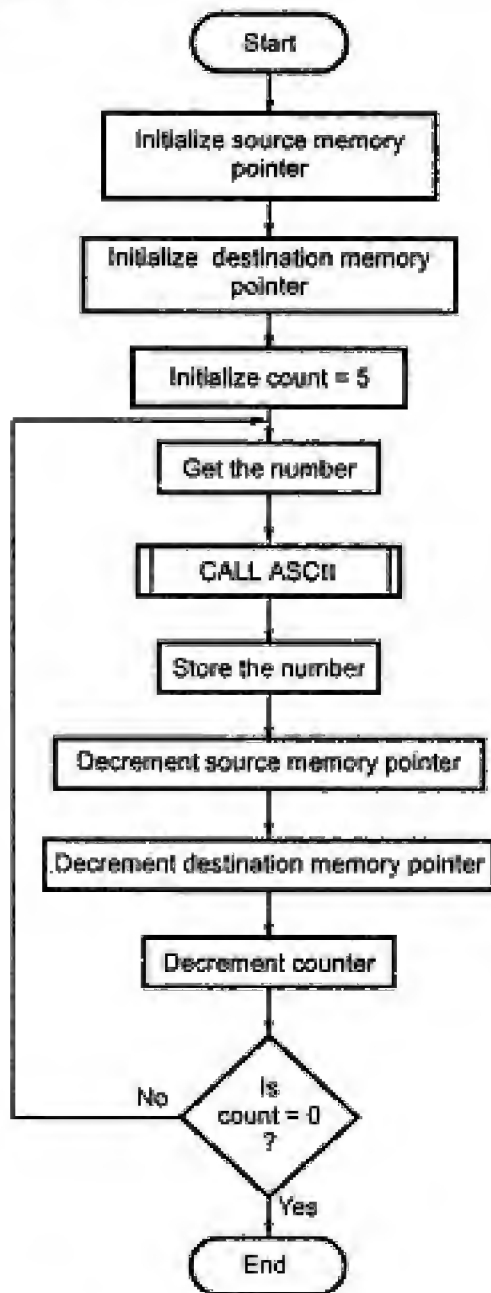
4.3.4 Binary to ASCII Code Conversion

The ASCII Code (American Standard Code for Information Interchange) is commonly used for communication. In such cases we need to convert binary number to its ASCII equivalent. It is a seven bit code. In this code number 0 through 9 are represented as 30 through 39 respectively and letters A through Z are represented as 41H through 5AH. Therefore, by adding 30H we can convert number into its ASCII equivalent and by adding 37H we can convert letter to its ASCII equivalent. Let us see the program for binary to ASCII code conversion.

Lab Experiment 56 : Find the ASCII character.

Statement : Write an assembly language program to convert the contents of the five memory locations starting from 2000H into an ASCII character. Place the result in another five memory locations starting from 2200H.

Flowchart



Sample problem

(2000H) = 1
 (2001H) = 2
 (2002H) = 9
 (2003H) = A
 (2004H) = B

Result (2200H) = 31
 (2201H) = 32
 (2202H) = 39
 (2203H) = 41
 (2204H) = 42

Subroutine Documentation :

Subroutine 'ASCII' converts a hexadecimal digit to ASCII.

Passing parameter : The digit is passed using accumulator.
 Return value : In the accumulator.
 Register used : Accumulator.
 Stack used : From 27FEH to 27FDH

Source program :

```

      LXI SP, 27FFH ; Initialize stack pointer
      LXI H, 2000H ; Source memory pointer
      LXI D, 2200H ; Destination memory pointer
      MVI C, 05H ; Initialize the counter
BACK:  MOV A, M ; Get the number
      CALL ASCII ; Call subroutine ASCII
      STAX D ; Store result
      INX H ; Increment source memory pointer
      INX D ; Increment destination memory pointer
      DCR C ; Decrement count by 1
      JNZ BACK ; if not zero, repeat
      HLT ; Stop program execution subroutine ASCII
ASCII: CPI, 0AH ; Check if number is 0AH
      JNC NEXT ; If yes goto next otherwise continue
      ADI 30H ;
      JMP LAST
NEXT:  ADI 37H
LAST:  RET ; Return to main program

```

4.3.5 ASCII Code to Binary Conversion

It is exactly reverse process to binary to ASCII conversion. Here, if ASCII code is less than 3AH then 30H is subtracted to get the binary equivalent and if it is in between 41H and 5AH then 37H is subtracted to get the binary equivalent of letter (A-F).

4.4 BCD Arithmetic**4.4.1 BCD Addition**

The addition of two BCD numbers can be best understood by considering the three cases that occur when two BCD digits are added.

Sum equals 9 or less with carry 0

Let us consider additions of 3 and 6 in BCD.

| | | |
|-----|---------|-------------|
| 6 | 0 1 1 0 | ← BCD for 6 |
| + 3 | 0 0 1 1 | ← BCD for 3 |
| 9 | 1 0 0 1 | ← BCD for 9 |

The addition is carried out as in normal binary addition and the sum is 1 0 0 1, which is BCD code for 9.

Sum greater than 9 with carry 0

Let us consider addition of 6 and 8 in BCD

$$\begin{array}{rcl}
 6 & 0110 & \leftarrow \text{BCD for 6} \\
 + 8 & 1000 & \leftarrow \text{BCD for 8} \\
 \hline
 14 & 1110 & \leftarrow \text{Invalid BCD number } (1110) > 9
 \end{array}$$

The sum 1 1 1 0 is an invalid BCD number. This has occurred because the sum of the two digits exceeds 9. Whenever this occurs the sum has to be corrected by the addition of six (0110) in the invalid BCD number, as shown below

$$\begin{array}{rcl}
 6 & 0110 & \leftarrow \text{BCD for 6} \\
 + 8 & 1000 & \leftarrow \text{BCD for 8} \\
 \hline
 14 & 1110 & \leftarrow \text{Invalid BCD number} \\
 + & 0110 & \leftarrow \text{Add 6 for correction} \\
 \hline
 \underbrace{0001}_1 & \underbrace{0100}_4 & \leftarrow \text{BCD for 14}
 \end{array}$$

After addition of 6 carry is produced into the second decimal position.

Sum equals 9 or less with carry 1

Let us consider addition of 8 and 9 in BCD

$$\begin{array}{rcl}
 8 & 1000 & \leftarrow \text{BCD for 8} \\
 + 9 & 1001 & \leftarrow \text{BCD for 9} \\
 \hline
 17 & 00010001 & \leftarrow \text{Incorrect BCD result}
 \end{array}$$

In this, case, result (0001 0001) is valid BCD number, but it is incorrect. To get the correct BCD result correction factor of 6 has to be added to the least significant digit sum, as shown.

$$\begin{array}{rcl}
 8 & 1000 & \leftarrow \text{BCD for 8} \\
 + 9 & 1001 & \leftarrow \text{BCD for 9} \\
 \hline
 17 & 00010001 & \leftarrow \text{Incorrect BCD result} \\
 + & 00000110 & \leftarrow \text{Add 6 for correction} \\
 \hline
 & 00010111 & \leftarrow \text{BCD for 17}
 \end{array}$$

Going through these three cases of BCD addition we can summarise the BCD addition procedure as follows :

1. Add two BCD numbers using ordinary binary addition.
2. If four-bit sum is equal to or less than 9, no correction is needed. The sum is in proper BCD form.

3. If the four-bit sum is greater than 9 or if a carry is generated from the four-bit sum, the sum is invalid.
4. To correct the invalid sum, add 0110_2 to the four-bit sum. If a carry results from this addition, add it to the next higher-order BCD digit.

The 8085 supports DAA (Decimal Adjust Accumulator) instruction for adjusting the result of addition to the BCD number. (See chapter 2 for DAA instruction).

Lab Experiment 57 : Add two 2-digit BCD numbers.

Statement : Add two 2 digit BCD numbers in memory location 2200H and 2201H and store the result in memory location 2300H.

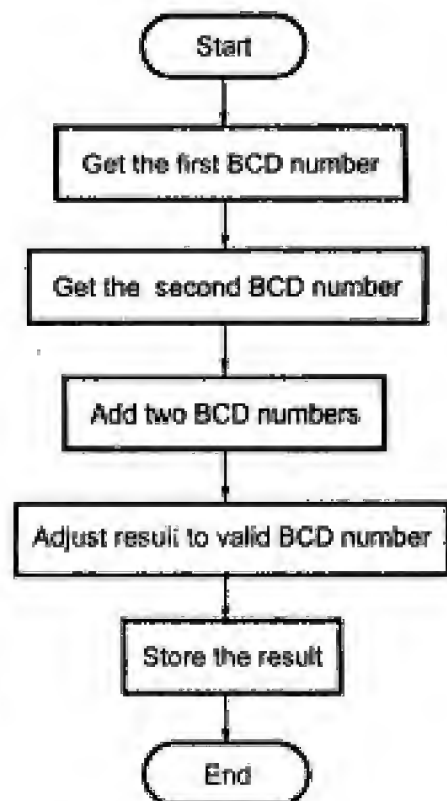
Sample problem

(2200H) = 39
 (2201H) = 45
 Result = (2300H) = 39 + 45
 = 7E + 6 = 84
 (lower nibble is greater than 9 so add 6)

Source program

```
LXI H, 2200H ; Initialize pointer
MOV A, M     ; Get the first number
INX H       ; Increment the pointer
ADD M       ; Add two numbers
DAA         ; Convert HEX to valid BCD
STA 2300H   ; Store the result
HLT         ; Terminate program
           ; execution
```

Flowchart



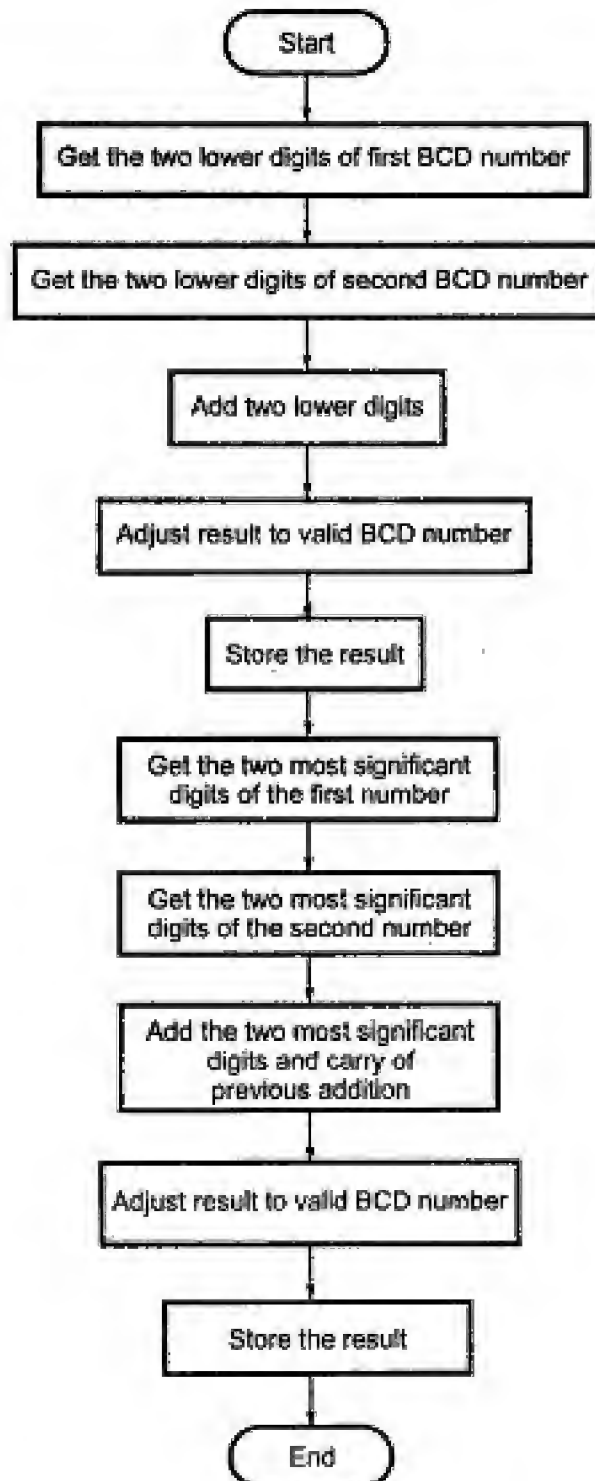
Lab Experiment 58 : Add two 4-digit BCD numbers.

Statement : Add two 4 digit BCD numbers in HL and DE register pairs and store result in memory locations, 2300H and 2301H. Ignore carry after 16 bit.

Sample problem

(HL) = 3629
 (DE) = 4738
Step 1 : 29 + 38 = 61 and auxiliary carry flag = 1
 ∴ add 06
 61 + 06 = 67

Flowchart



Step 2 : $36 + 47 + 0 \text{ (carry of LSB)} = 7D$
 Lower nibble of addition is greater than 9
 so add 6.
 $7D + 06 = 83$
 \therefore Result = 8367

Source program

```
MOV A, L      ; Get lower 2 digits of no. 1
ADD E         ; Add two lower digits
DAA           ; Adjust result to valid BCD
STA 2300H     ; Store partial result
MOV A, H      ; Get most significant 2 digits of no. 2
ADC D         ; Add two most significant digits
DAA           ; Adjust result to valid BCD
STA 2301H     ; Store partial result
HLT           ; Terminate program execution.
```

4.4.2 BCD Subtraction

When two BCD numbers are subtracted we can use DAA instruction for adjusting result to the BCD. Therefore, the subtraction of BCD number is carried out using 10's complement or 100's complement method.

The 10's complement of a decimal number is equal to the 9's complement plus 1 and the 100's complement of a decimal number is equal to the 99's complement plus 1. The 99's complement of a number can be found by subtracting the number from 99. The steps for 100's complement BCD subtraction are as follows :

- Find the 100's complement of subtrahend.
- Add two numbers using BCD addition.

Let us see the program for subtraction of two BCD numbers.

Lab Experiment 59 : Subtraction of two BCD numbers.

Statement : Subtract the BCD number stored in E register from the number stored in the D register.

Source program

```
MVI A, 99H
SUB E         ; Find the 99's complement of subtrahend
INR A         ; Find 100's complement of subtrahend
ADD D         ; Add minuend to 100's complement of subtrahend
DAA           ; Adjust for BCD
HLT           ; Terminate program execution
```

Lab Experiment 60 : Multiply two 2-digit BCD numbers.

Statement : Write an assembly language program to multiply 2 BCD numbers.

Source program

```
MVI C, Multiplier      ; Load BCD multiplier
MVI B, 00               ; Initialize counter
LXI H, 0000H           ; Result = 0000
MVI E, multiplicand     ; Load multiplicand
```

```

      MVI D, 00H           ; Extend to 16-bits
BACK : DAD D              ; Result ← Result + Multiplicand
      MOV A, L             ; [ Get the lower
      ADI, 00H            ;   byte of the result
      DAA                 ;   Adjust it to BCD and
      MOV L, A             ;   store it]
      MOV A, H             ; [ Get the higher
      ACI, 00H            ;   byte of the result
      DAA                 ;   Adjust it to BCD and
      MOV H, A             ;   store it]
      MOV A, B             ; [ Increment
      ADI 01 H            ;   counter
      DAA                 ;   adjust it to BCD and
      MOV B, A             ;   store it]
      CMP C                ; Compare if count = multiplier
      JNZ BACK            ; if not equal repeat
      HLT                 ; Stop

```

Review Questions

1. What do you mean by Looping, Counting and Indexing ? Explain with the help of example.
2. Explain the need of software timers.
3. Explain how software delays can be implemented using counters.
4. List the common code conversions required in the microprocessor systems.
5. Explain BCD to Binary code conversion technique and write 8085 assembly language program for the same.
6. Explain Binary to BCD code conversion technique and write 8085 assembly language program for the same.
7. Explain BCD to Seven segment code conversion technique and write 8085 assembly language program for the same.
8. Explain Binary to ASCII code conversion technique and write 8085 assembly language program for the same.
9. Explain ASCII to Binary code conversion technique and write 8085 assembly language program for the same.
10. Explain the procedure for addition of two BCD numbers.
11. Explain the procedure for subtraction of two BCD numbers.

□□□

Stacks and Subroutines

The stack is a part of read/write memory that is used for temporary storage of binary information during the execution of a program. The binary information is basically the intermediate results and the return address in case of subroutine calls.

1. For the application programs, the internal memory of the microprocessor (registers) is not sufficient to store the intermediate results. These intermediate results can be stored temporarily on the stack and can be referred back when required.
2. A subroutine is a group of instructions, performs a particular subtask which is executed number of times. It is written separately. The microprocessor executes this subroutine by transferring program control to the subroutine program. After completion of subroutine program execution, the program control is returned back to the main program. The use of subroutines is a very important technique in designing software for microprocessor systems because it eliminates the need to write a subtasks repeatedly; thus it uses memory more efficiently. For implementation of subroutine technique, it is necessary to define stack. In the stack, the address of the instruction in the main program which follows the subroutine call is stored.

5.1 Concept of Stack and Subroutines

This section explains the concept of stack and subroutines in detail.

5.1.1 Stack

The stack is a portion of read/write memory set aside by the user for the purpose of storing information temporarily. When the information is written on the stack, the operation is called PUSH. When the information is read from stack, the operation is called POP.

The microprocessor stores the information, much like stacking plates. Using this analogy of stacking plates it is easy to illustrate the stack operation.

Fig. 5.1 shows the stacked plates. Here, we realize that if it is desired to take out the first stacked plate we will have to remove all plates above the first plate in the reverse

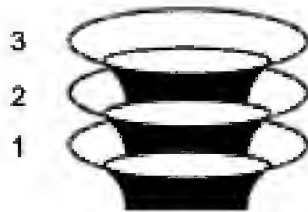


Fig. 5.1 Stacked plates

order. This means that to remove first plate we will have to remove the third plate, then the second plate and finally the first plate. This means that, the first information pushed on to the stack is the last information popped off from the stack. This type of operation is known as a first in, last out (FILO). This stack is implemented with the help of special memory pointer register. The special pointer register is called the **stack pointer**. During PUSH and POP operation, stack

pointer register gives the address of memory where the information is to be stored or to be read. The stack pointer's contents are automatically manipulated to point to stack top. The memory location currently pointed by stack pointer is called **top of stack**.

The stack pointer SP, is a 16-bit register in the 8085A which is manipulated by the microprocessor's control section, during stack related instructions.

5.1.1.1 Stack Related Instructions

The 8085A supports following stack related instructions :

LXI SP, data (16) :

It initializes stack pointer with 16-bit address.

SPHL :

It copies the contents of HL register pair into the stack pointer.

PUSH rp :

It is used to write 16-bit data in the stack.

POP rp :

It is used to read 16-bit data from the stack.

CALL addr :

It transfers the program control to the subroutine program after storing the return address in the stack.

RET :

It reads the return address from the stack and transfers the program control back to the instruction following the CALL.

INX SP :

It increments the contents of stack pointer by one.

DAD SP :

It adds the contents of stack pointer into the contents of HL register pair and stores the result in the HL register pair.

XTHL :

It exchanges the contents of memory location pointed by the stack pointer with the contents of L register and the contents of the next memory location with the contents of H register.

Now we will see the detail operation and the use of these instructions.

5.1.1.2 Detail Operation and the use of Stack Related Instructions**LXI SP, Data and SPHL : Initializes Stack Pointer.**

Before execution of any stack related instruction, stack pointer must be initialized with a valid memory address. The stack pointer can be initialized by two ways.

1. Direct way : LXI SP, data (16-bit) ; Loads 16-bit data into SP
2. Indirect way : LXI H, data (16-bit) ; Loads 16-bit data into HL
SPHL ; Loads the contents of HL into SP

Normally, the stack pointer is initialized by the direct way. When a programmer wishes to set the stack pointer to a value that has been computed by the program, indirect way is used. The computed value is placed in H and L and the contents of HL register pair then moved into the stack pointer.

Note :

1. The stack pointer can be initialized anywhere in the read/write memory map. However, as a general practice, the stack pointer is initialized at the highest Read/Write memory location so that it will be less likely to interfere with a program.
2. Since the 8085A's stack pointer is decremented before data is written to the stack, the stack pointer can actually be initialized to a value one higher than the highest read/write memory location available.

PUSH and POP : Temporarily stores the contents of register pair and program internal status word, and retrieves when required.

When programmer realizes the shortage of the registers, he stores the present contents of the registers in the stack with the help of PUSH instruction and then uses the registers for other function. After completion of other function programmer loads the previous contents of the register from the stack with the help of POP instruction.

PUSH Operation : In PUSH operation, 16-bit data is stored on the stack. This 16-bit data is stored in two operations. In the first operation, stack pointer is decremented by one and then the higher byte of the 16-bit data is stored at the memory location pointed by stack pointer. In the second operation, stack pointer is again decremented by one and then the lower byte of the 16-bit data is stored at the memory location pointed by stack pointer. The Fig. 5.2 shows steps involved in the PUSH operation.

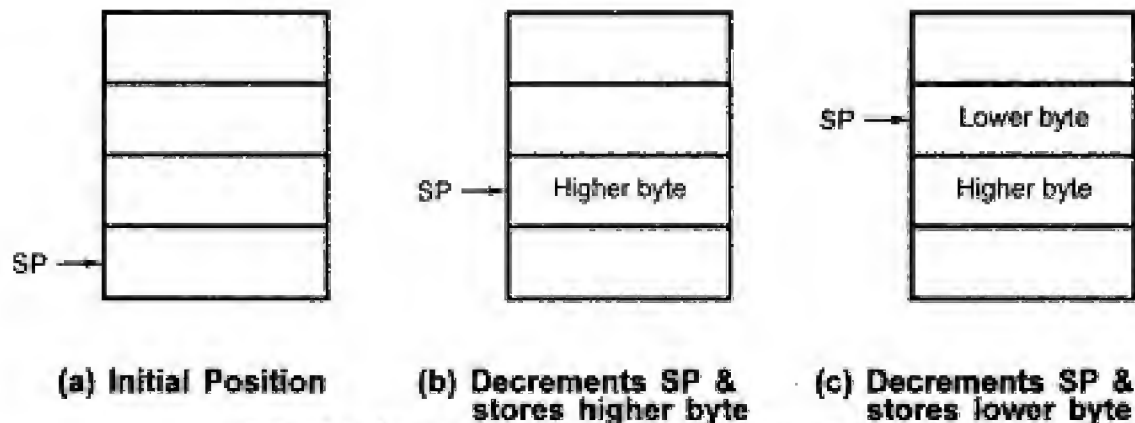


Fig. 5.2 Steps Involved in PUSH Operation

Example 1 : The contents of BC register pair are 1020H and contents of stack pointer are 27FFH. Then after execution of PUSH B instruction the stack contents are as shown in the Fig. 5.3.

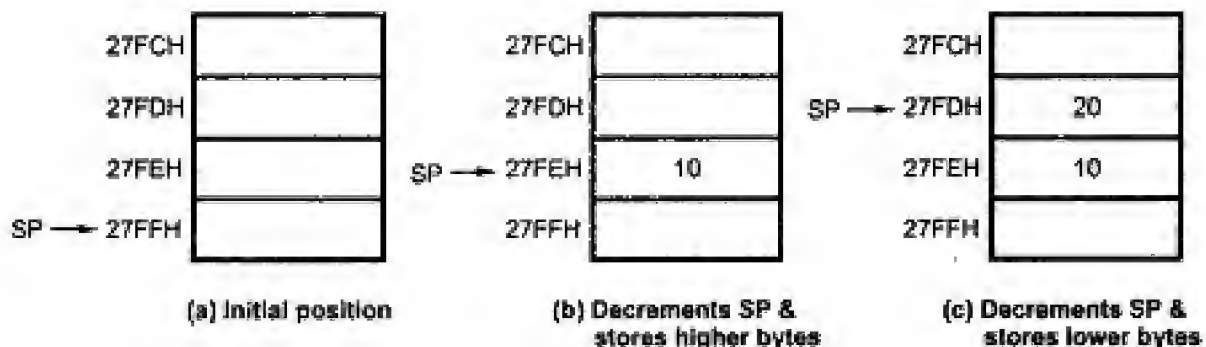


Fig. 5.3 Steps Involved in PUSH Operation with example

POP Operation : In POP operation, 16-bit data is read from the stack. This 16-bit data is read in two operations. In the first operation, the contents from the memory location pointed by stack pointer are loaded into lower byte of register pair and then the stack pointer is incremented by one. In the second operation, the contents from the

memory location pointed by stack pointer are loaded into higher byte of register pair and then the stack pointer is incremented by one. Fig. 5.4 shows steps involved in the POP operation.

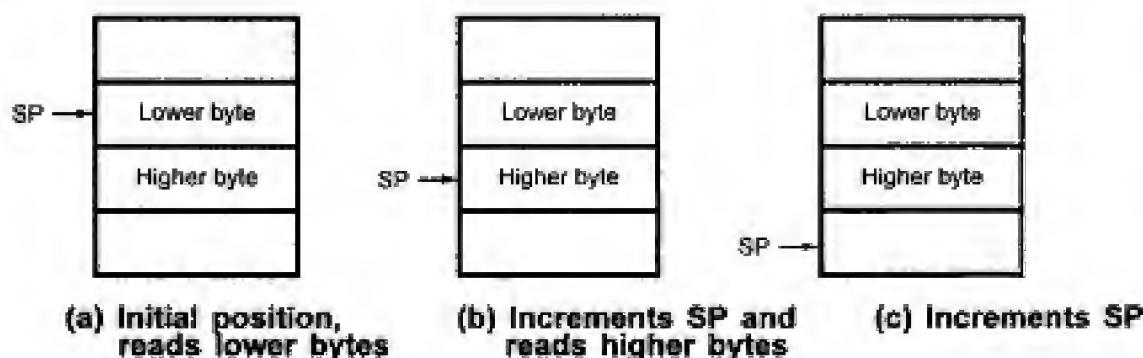


Fig. 5.4 Steps Involved in POP Operation

Example 2 : If the initial contents of stack and contents of stack pointer are as shown in Fig. 5.5 (a) then after execution POP B contents of BC register will be 3040 (B = 30H and C = 40H).

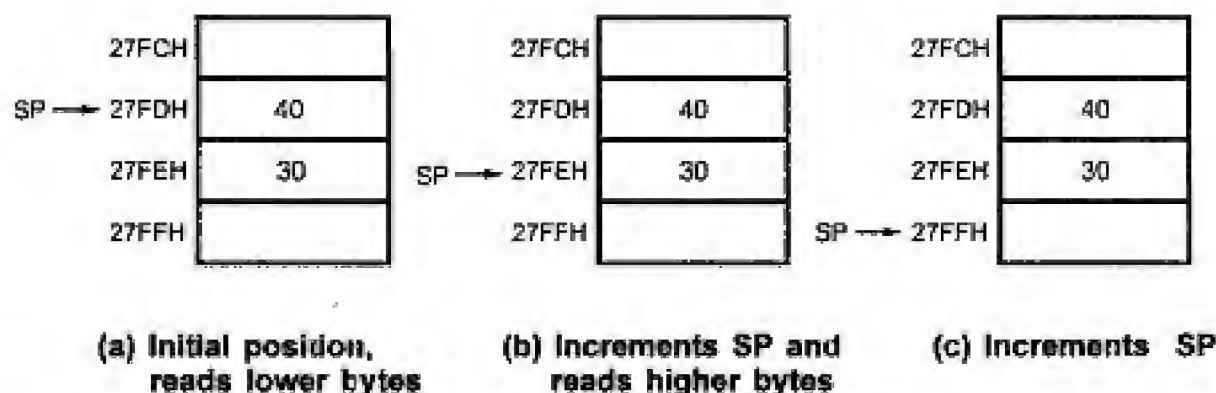


Fig. 5.5 Steps Involved in POP operation with example

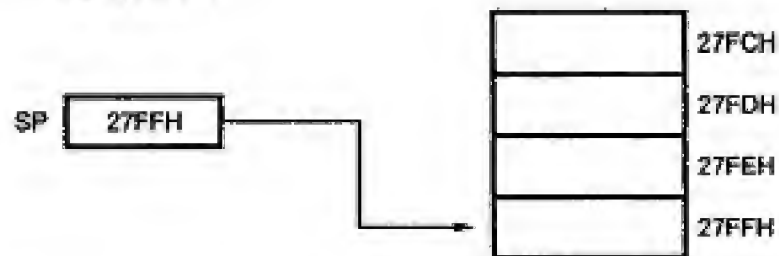
Example 3 : Let us consider the following program

```
LXI SP, 27FFH
LXI B, 2030H
LXI D, 4045H
PUSH D
PUSH B
MOV A, C
ADD E
MOV D, A
POP B
POP D
```

We will see the contents of stack and stack pointer after execution of each instruction in the above program.

Instruction 1 :

LXI SP, 27FFH

**Instruction 2 :**

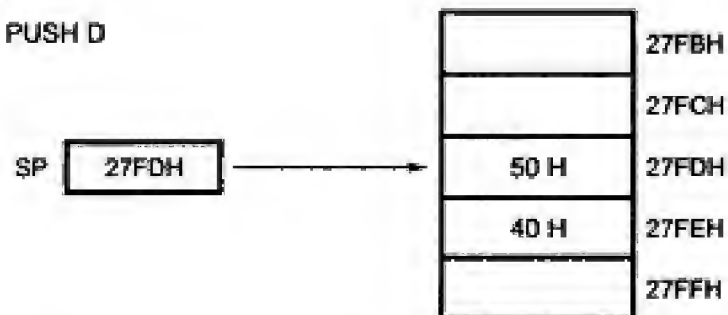
LXI B, 2030H

**Instruction 3 :**

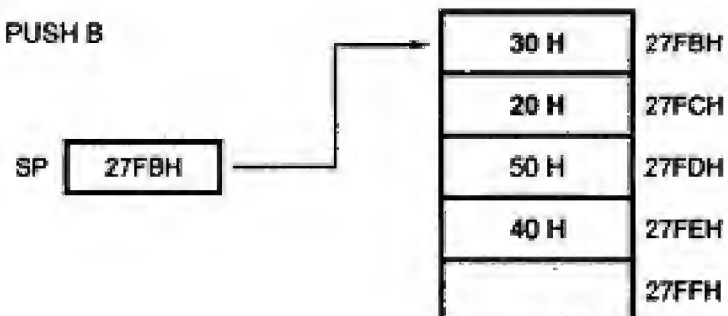
LXI D, 4050H

**Instruction 4 :**

PUSH D

**Instruction 5 :**

PUSH B



Instruction 6 :

MOV A, C

A 30 H

Instruction 7 :

ADD E

A 80 H

Instruction 8 :

MOV D, A

D 80 H

Instruction 9 :

POP B

SP 27FDH

B

20 H

C

30 H

| | |
|------|-------|
| 30 H | 27FBH |
| 20 H | 27FCH |
| 50 H | 27FDH |
| 40 H | 27FEH |
| | 27FFH |

Instruction 10 :

POP D

SP 27FFH

D

40 H

E

50 H

| | |
|------|-------|
| 20 H | 27FBH |
| 30 H | 27FCH |
| 50 H | 27FDH |
| 40 H | 27FEH |
| | 27FFH |

This program shows us how registers can be used for more than one purpose. This program initializes the stack pointer at 27FFH and stores the original contents of BC and DE register pairs in the stack. Now registers B, C, D and E are free to use for intermediate

calculations. Once these calculations are over, we can get back the original contents of B, C, D and E registers from stack by POP B and POP D instructions. We know that stack works in FILO fashion. Therefore, the sequence of getting the contents back from the stack should be exactly in the reverse order that of the sequence of storing the contents in the stack. In this example the contents of BC register pair are stored after storing the contents of DE register pair in the stack. But while getting the contents of register pairs from the stack, first BC register pair contents are retrieved and then DE register pair contents are retrieved.

➡ **Example 5.1 :** Exchange the contents of BC and DE registers without using any other MPU general purpose register.

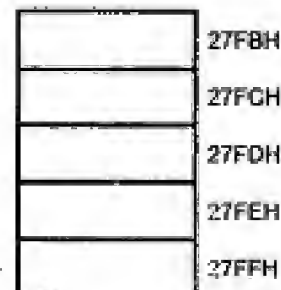
Solution : LXI SP, 27FFH
 PUSH B
 PUSH D
 POP B
 POP D

We will see the contents of stack and stack pointer after execution of each instruction in the above program.

Instruction 1 :

LXI SP, 27FFH

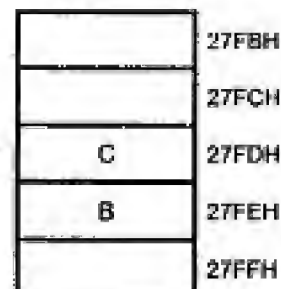
SP 27FFH



Instruction 2 :

PUSH B

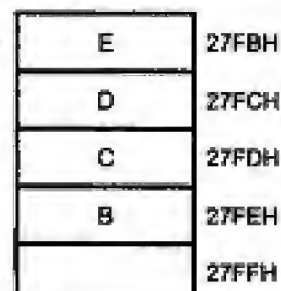
SP 27FDH



Instruction 3 :

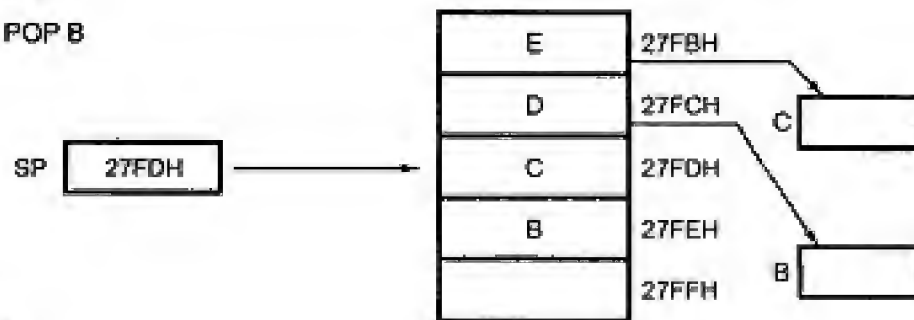
PUSH D

SP 27FBH



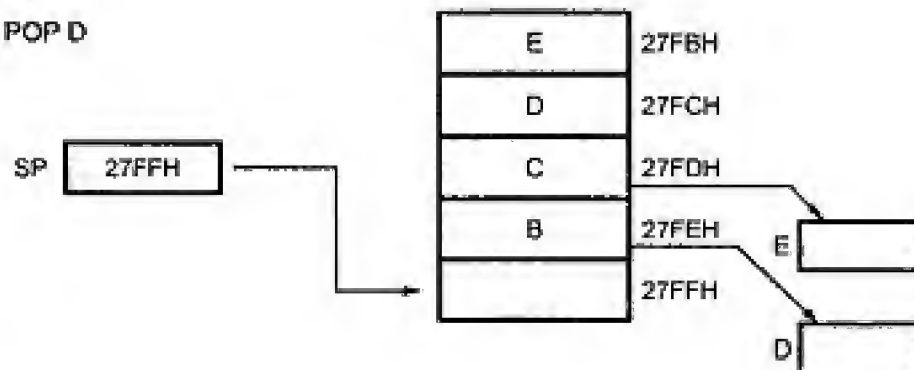
Instruction 4 :

POP B



Instruction 5 :

POP D



This program shows us that if you reverse the order of retrieving the contents from the stack we will not get the original contents of the register pairs. Here, the contents of DE and BC register pairs will be exchanged after execution of the program.

➡ **Example 5.2 :** Write a program to load the flag register contents in C register.

Solution : We know that 8085 does not provide any instruction to transfer the contents of flag register to any general purpose register. Therefore, to load flag register contents into any register, it is necessary to use stack. Following program explains how to use stack to load flag register contents in the C register.

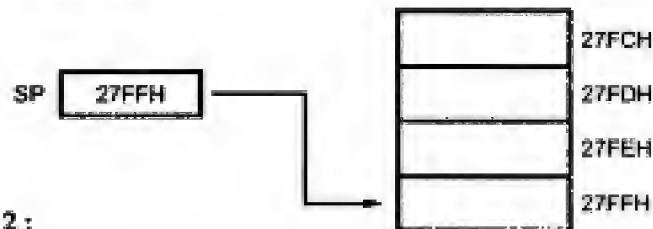
Program :

```
LXI SP, 27FFH
PUSH PSW
POP B
```

We will see the contents of stack and stack pointer after execution of each instruction in the program.

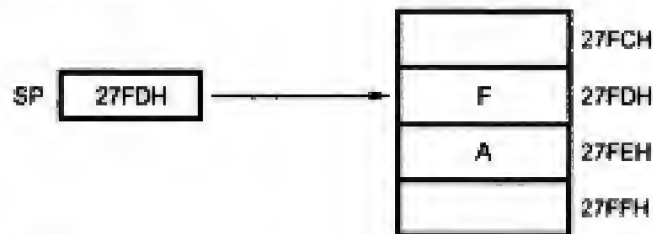
Instruction 1 :

LXI SP, 27FFH



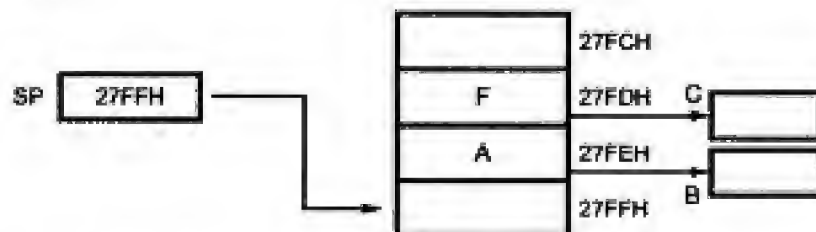
Instruction 2 :

PUSH PSW



Instruction 3 :

POP B



CALL Address and RET : Implements Subroutines

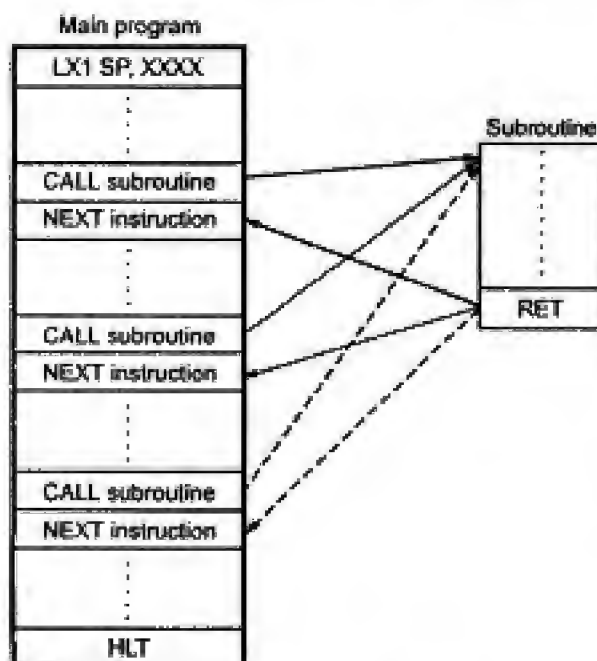


Fig. 5.6 Program flow

Whenever we need to use a group of instructions several times throughout a program there are two ways to avoid rewriting of the group of instructions. One way is to write the group of instructions as a separate subroutine. We can then just CALL the subroutine whenever we need to execute that group of instructions. For calling the subroutine we have to store the return address on the stack. This process takes some time. If the group of instructions is bit enough then this overhead time and execution time are comparable. In such cases, it is not desirable to write subroutines. For these cases, we can use macros. Macro is also a group of

instructions. Each time we "CALL" a macro in our program, the assembler will insert the defined group of instructions in place of the "CALL". An important point here is that the assembler generates machine codes for the group of instructions each time macro is called. So there is not overhead time involved in calling and returning from a subroutine. The disadvantage of macro is that it generates in line code each time when the macro is called which takes more memory.

5.1.2 Subroutines

From the previous discussions, we know that the subroutine is a group of instructions stored as a separate program in the memory and it is called from the main program whenever required.

The 8085A microprocessor has two instructions to implement subroutines: **CALL** and **RET**. **CALL** instruction is used to call a subroutine in the main program and **RET** instruction is the last instruction in the subroutine to return it back to the main program. The **CALL** instruction saves the address of the instruction following it and then transfers the program control to the first instruction in the subroutine. When subroutine execution is completed the **RET** instruction reads the return address from the stack and transfers control back to the instruction following the **CALL**.

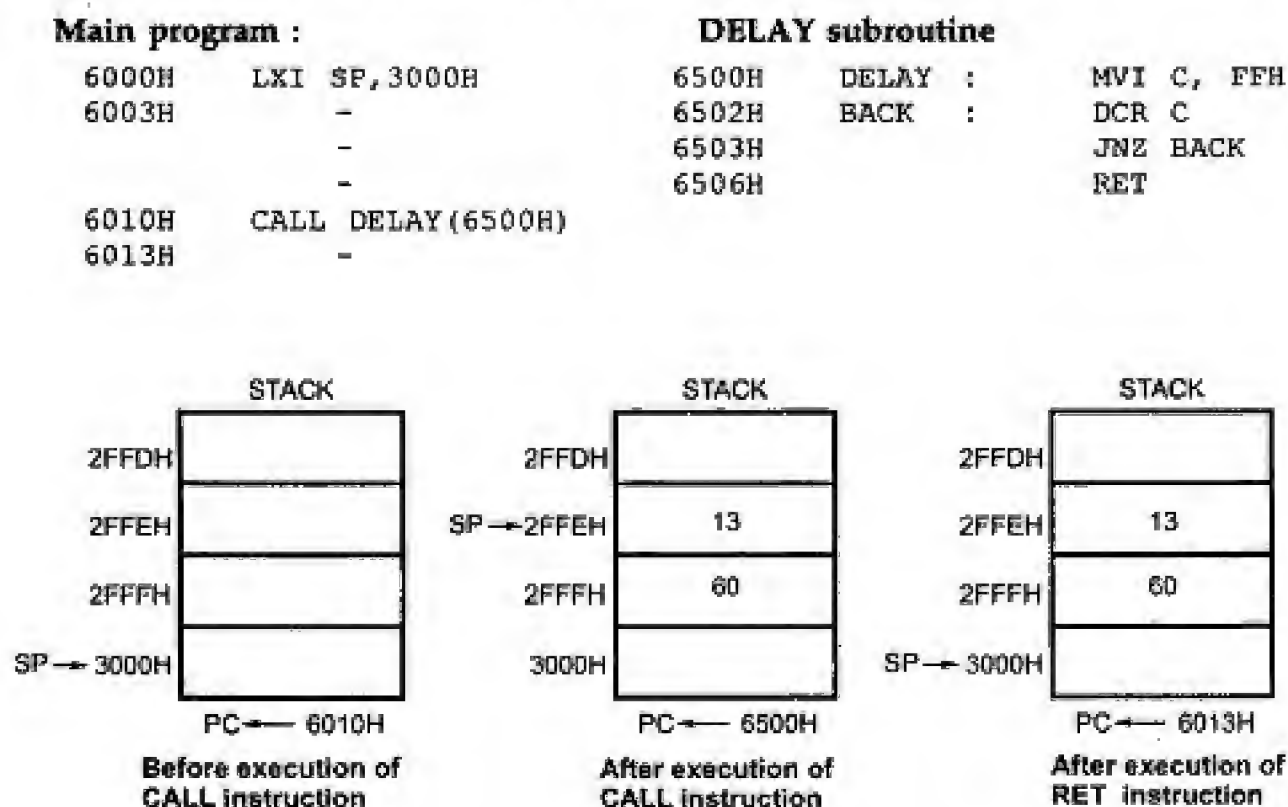


Fig. 5.7 Details in the execution of CALL and RETURN instructions

Here, the main program initializes stack pointer at 3000H memory location and executes instructions in sequence till the execution of CALL instruction. After execution of CALL instruction program control is transferred to the delay subroutine stored at memory address 6500H. Before transfer of control to the subroutine the address of the instruction (6013H) which is after the CALL instruction, is stored in the stack. At the end of delay subroutine RET instruction is executed, which reads the return address (6013H) from the stack and transfers the program control to the instruction which is after CALL instruction.

Conditional Call and Return Instructions :

In addition to the unconditional CALL and RET instructions, the 8085A instruction set includes eight conditional CALL instructions and eight conditional RET instructions. These conditions are checked by reading the status of respective flags. If the condition associated with the conditional CALL is not met, the instruction following the CALL is executed. If the condition is met, the program counter contents are saved on the stack, and the address contained in the CALL instruction is loaded into program counter. The number of machine cycles and T-states required by a conditional CALL depends on whether or not the condition is satisfied. When the condition is not satisfied, two machine cycles with a total of nine T-state are required to fetch, decode and execute the instruction. When the condition is satisfied, five machine cycles with 18 T-states are required.

Conditional CALL Instructions

| Instruction code | Description | Condition for CALL |
|------------------|---------------------|--------------------|
| CC | Call on carry | CY = 1 |
| CNC | Call on not carry | CY = 0 |
| CP | Call on positive | S = 0 |
| CM | Call on minus | S = 1 |
| CPE | Call on parity even | P = 1 |
| CPO | Call on parity odd | P = 0 |
| CZ | Call on zero | Z = 1 |
| CNZ | Call on not zero | Z = 0 |

Table 5.1 Conditional calls

In case of a conditional return instruction, the sequence returns to the main program if the condition is met otherwise, the sequence in the routine is continued.

| Instruction code | Description | Condition for RET |
|------------------|-----------------------|-------------------|
| RC | Return on carry | CY = 1 |
| RNC | Return on not carry | CY = 0 |
| RP | Return on positive | S = 0 |
| RM | Return on minus | S = 1 |
| RPE | Return on parity even | P = 1 |
| RPO | Return on parity odd | P = 0 |
| RZ | Return on zero | Z = 1 |
| RNZ | Return on not zero | Z = 0 |

Table 5.2 Conditions for return

5.2 Parameters Passing Techniques

We often want a subroutine to process some data or address variable from the main program. For processing it is necessary to pass these address variables or data, usually referred to as passing parameters to the subroutine. There are four ways to pass parameters to and from the subroutine :

1. Using registers
2. Using general memory
3. Using pointers
4. Using stack

5.2.1 Passing Parameters using Registers

The data, to be passed is stored in the registers and these registers are accessed in the subroutine to process the data. In this technique, the main program loads internal registers with appropriate values before calling the subroutine and subroutine then obtains these values by referring pre-defined registers, as shown in the following example.

Example :

Passing Parameters Using Registers

```
; Main program
.
.
MVI C, 08H      ; Data to be passed is loaded in the
                  ; register
CALL SUB1
MOV A, D         ; Main program accesses result from
                  ; register.
.
.
```

```

; subroutine
SUB1:  MOV B, C      ; Subroutine accesses the data from C
                        ; register
      .
      MOV D, A      ; Stores result in D register.
      RET

```

5.2.2 Passing Parameters using Memory

For the cases where we have to pass few parameters to and from a subroutine, registers are a convenient way to do it. However in cases where we need to pass a large number of parameters to subroutine we use memory. This memory may be a dedicated section of general memory or a part of stack. In this technique, the main program loads the pre-defined memory locations with appropriate values before calling the subroutine and subroutine then obtains these values by referring these predefined memory locations, as shown in the following example.

Example :

Passing Parameters Using General Memory

```

; Main program
      LXI H, 2200H ; Initialize memory pointer
      MVI M, 50H  ; Load data into memory
      .
      .
      CALL SUB1
      LDA 2300H   ; Main program accesses result from
                  ; memory location 2300H.
      .
; Subroutine
SUB1:  LDA 2200H   ; Subroutine accesses the data from
                  ; memory location.
      .
      .
      STA 2300H   ; Stores result in memory location
                  ; 2300H
      RET

```

The subroutine program stores the generated results in the memory locations before execution of RET. The main program accesses results from the these memory locations for further processing.

5.2.3 Passing Parameters using Pointers

In this technique, the main program stores the parameters to be passed in the memory, usually in the consecutive memory locations. Then it loads the internal register pair or pre-defined memory locations with the starting address of the parameter list. The subroutine obtains parameter list by accessing it in sequence from the given address.

Example :

Passing Parameters Using Pointers

```

; Main program

```

```

    LXI H, 2200H      ; Initialize memory
    MOV M, A
    INX H
    MOV M, B
    :
    :
    INX H
    MVI M, 30H
    :
    SHLD 3000H        ; Store memory pointer
    CALL SUB1
    :
    :
; Subroutine
SUB1: LHLD 3000H      ; Subroutine access pointer (address) to data
      MOV A, M
      :
      :
      RET

```

5.2.4 Passing Parameters using Stack

The stack can be used to pass parameters. To pass parameters to the subroutine using stack, it is necessary to push them on the stack before the call for the subroutine in the main program. The instructions in the subroutine read these parameters from the stack. Whenever the stack is used to pass parameters, it is very important to keep track of what is pushed on the stack and where the stack pointer points all the time in the program.

Example :

Passing Parameters Using Stack

```

; Main program
    LXI B, 1020H      ; Load BC register pair with 1020H
    PUSH B            ; Store BC register pair on stack.
    CALL SUB1
    :
; Subroutine
SUB1
    POP H              ; Store the return address into HL register
                       ; pair.
    POP B              ; The subroutine accesses data from stack using
                       ; BC register pair.
    PCHL               ; Here it is not possible to use return
                       ; instruction, since stack pointer not pointing
                       ; the return address. But the address is available
                       ; in HL. Thus the PCHL instruction is used to
                       ; transfer program control to the main program.

```

Example : Write a program to exchange the higher and lower nibble of ten 8-bit numbers stored from location 2200H. Make use of subroutine and explain different parameter passing techniques.

Solution :

a) Parameter passing using register A

Main program :

```

        LXI SP, 27FFH    ; Initializes stack pointer
        LXI H, 2200H     ; Initializes memory pointer
        MVI C, 0AH       ; Initializes counter
BACK:   MOV A, M          ; Stores number (passing parameter)
                           ; in A register
        CALL EXCHANGE    ; Calls subroutine exchange
        MOV M, A          ; Stores the result from register A
        INX H             ; Increments memory pointer
        DCR C             ; Decrements counter
        JNZ BACK          ; If not zero, repeat
        HLT              ; Stop

```

Subroutine program :

```

EXCHANGE: RLC
          RLC
          RLC
          RLC              ; Rotate 4 times left to exchange
                           ; the nibbles.
          RET              ; Return to the main program

```

b) Passing parameters using memory location

Main program :

```

        LXI SP, 27FFH    ; Initializes stack pointer
        LXI H, 2200H     ; Initializes memory pointer
        MVI C, 0AH       ; Initializes counter
BACK :   MOV A, M          ; Gets the number
        STA 2300H         ; Stores the number (parameter) at
                           ; 2300H
        CALL EXCHANGE    ; Calls subroutine exchange
        LDA 2300H         ; Gets the result
        MOV M, A          ; Stores the result
        INX H             ; Increments memory pointer
        DCR C             ; Decrements counter
        JNZ BACK          ; If not zero, repeat
        HLT              ; Stop

```

Subroutine program :

```

EXCHANGE: LDA 2300H       ; Gets the parameter
          RLC
          RLC
          RLC
          RLC              ; Rotate 4 times left to exchange
                           ; the nibbles
          STA 2300H       ; Store the result
          RET              ; Return to main program

```

c) Passing parameters using pointer

```

        LXI SP, 27FFH    ; Initializes stack pointer
        LXI H, 2200H     ; Initializes memory pointer
        MVI C, 0AH       ; Initializes counter
BACK:   SHLD 2300H        ; Store pointer to passing
                        ; parameter
        CALL EXCHANGE     ; Calls subroutine exchange
        INX H             ; Increments memory pointer
        DCR C             ; Decrements counter
        JNZ BACK          ; If not zero, repeat
        HLT              ; Stop

```

Subroutine program :

```

EXCHANGE:  LHLD 2300H      ; Gets pointer to parameter
           MOV A, M        ; Gets number
           RLC
           RLC
           RLC
           RLC             ; Rotate 4 times left to exchange
                        ; the nibbles
           MOV M, A        ; Stores the result
           RET             ; Return to main program.

```

d) Passing parameters using stack.

```

        LXI SP, 27FFH    ; Initializes stack pointer
        LXI H, 2200H     ; Initializes memory pointer
        MVI C, 0AH       ; Initializes counter
BACK :   MOV A, M         ; Gets the number
        PUSH PSW          ; Stores number in stack as a
                        ; parameter
        CALL EXCHANGE     ; Calls subroutine exchange
        MOV M, A          ; Stores the result
        INX H             ; Increments memory pointer
        DCR C             ; Decrements counter
        JNZ BACK          ; If not zero repeat
        HLT              ; Stop

```

Subroutine program :

```

EXCHANGE:  POP H           ; Stores return address in HL
           POP PSW        ; Gets the number in accumulator
           RLC
           RLC
           RLC
           RLC             ; Rotate 4 times left to exchange
                        ; the nibbles
           PCHL           ; Return to main program.

```

5.3 Subroutine Documentation

Subroutine program must provide enough information so that other users can utilize the subroutine without having to examine its internal structure. So along with subroutine program it is necessary to give the following guidelines.

1. Description of the purpose of the subroutine
2. A list of passing parameters
3. Return value
4. Registers and memory and memory locations used
5. Sample code

If these guidelines are followed while writing the subroutine then the subroutine can be easily used as a library function in other applications if required.

5.4 Advanced Subroutine Concepts

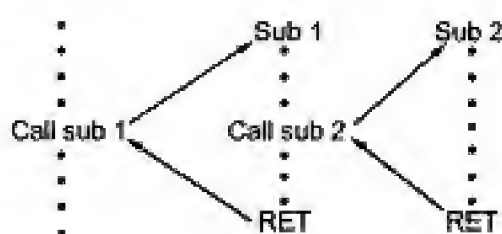


Fig. 5.8 Transfer of control with nested subroutines

When one subroutine calls another subroutine to complete a particular task, the operation is called **nesting**. The second subroutine may in turn call a third subroutine and so on each successive CALL without an intervening return creates an additional level of nesting. These routines are called **nested subroutines**. Fig. 5.8 shows the transfer of control with nested subroutines.

Nested subroutines are commonly classified as :

- Re-entrant Subroutine
- Recursive Subroutine

5.4.1 Re-entrant Subroutine

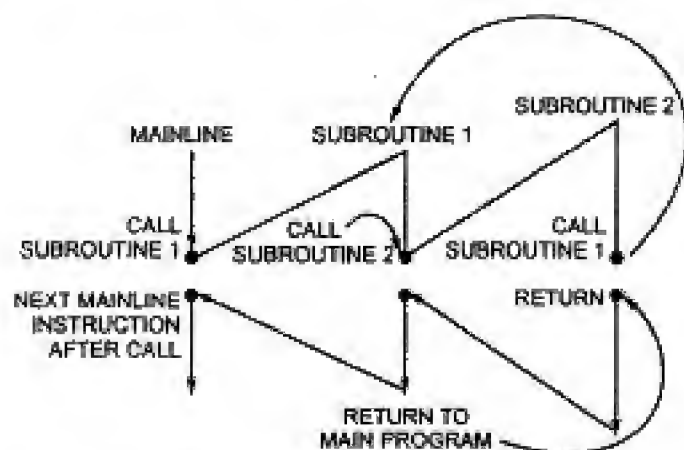


Fig. 5.9 Flow of program execution for re-entrant subroutine

In some situations it may happen that subroutine1 is called from main program, subroutine2 is called from subroutine1 and subroutine1 is again called from subroutine2. In this situation program execution flow re-enters in the subroutine1. This type of subroutines are called **re-entrant subroutines**. The flow of program execution for re-entrant subroutine is shown in Fig. 5.9.

5.4.2 Recursive Subroutine

A recursive subroutine is a subroutine which calls itself. Recursive subroutines are used to work with complex data structures called trees. If the subroutine is called with N (recursion depth) = 3, then the N is decremented by one after each subroutine CALL and the subroutine is called until $N = 0$. Fig. 5.10 shows the flow diagram and pseudo code for recursive subroutine.

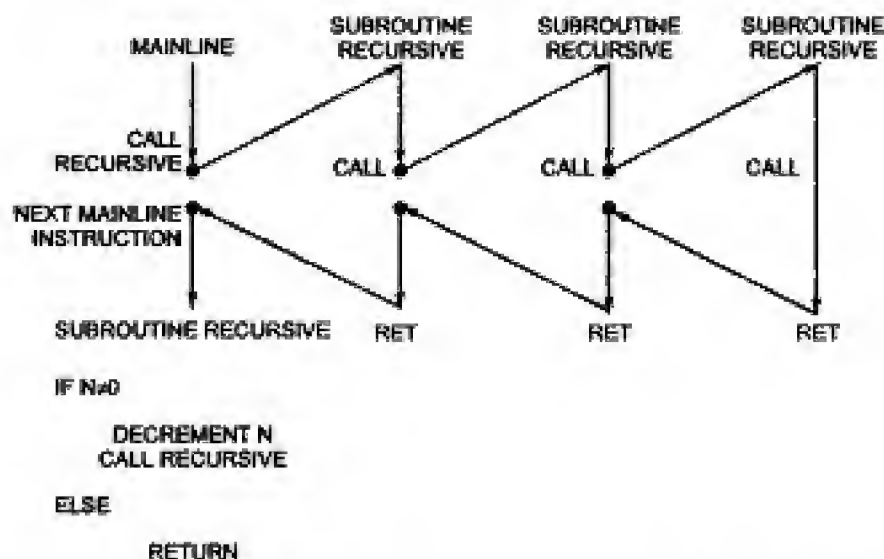


Fig. 5.10 Flow diagram and pseudo code for recursive subroutine

Limitations :

The level of subroutine nesting cannot exceed that supported by the available stack memory in the system. Since the stack pointer is usually initialized to the highest available read/write memory location, and the stack grows toward lower addresses, it is possible that too many PUSH or CALL operations without a sufficient number of intervening POP or RET operations result in the overflow of stack. For example, assume that read/write memory allocated in the system for stack has address range from 2000H to 27FFH. In this case, if stack grows beyond 2000H then it is called as overflow of stack. Care must be taken in memory allocation and program design to prevent this.

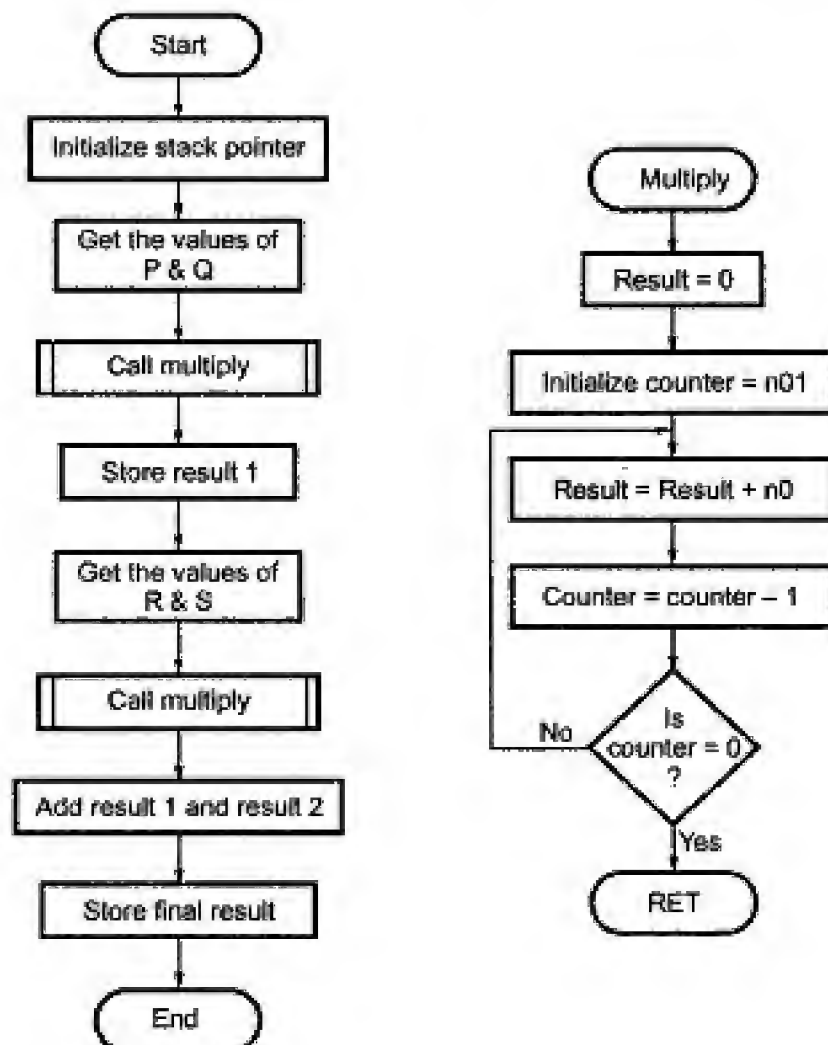
Lab Experiment 61 : Solve given equation.

Statement : Write a program to solve following equation

$$X = (P \times Q) + (R \times S)$$

Assume P , Q , R and S numbers are stored at 2200H, 2201H, 2202H and 2203H memory locations respectively. Store the result in memory locations 2204H and 2205H. Make use of subroutine.

Flowchart :



Main program :

```

LXI SP, 27FFH ; Initialize stack pointer
LXI H, 2200H ; Initialize memory pointer
CALL MULTIPLY ; Call subroutine multiply
SHLD 2204H ; Store the result 1
LXI H, 2202H ; Set memory pointer for next two numbers
CALL MULTIPLY ; Call subroutine multiply
XCHG ; Save result in DE register pair
LHLD 2204H ; Get the result 1
DAD D ; Add result 1 and result 2
SHLD 2204H ; Store final result
HLT ; Stop
  
```

Multiply subroutine :

```

MULTIPLY: MOV C, M ; Initialize number1 as a counter
          MVI D, 00H ;
  
```

```
                INX H           ;  
                MOV E, M       ; Get number 2  
                LXI H, 0000H    ; Result = 0  
BACK :          DAD D          ; Result = Result + number2  
                DCR C           ; Decrement counter  
                JNZ BACK       ; If not zero, repeat  
                RET            ; Return to main program
```

Review Questions

1. What is stack ? Explain the use of the stack, and stack pointer and how they are affected by instructions such as PUSH, POP, CALL and RET.
2. What is subroutine ? How is it useful ? Explain the use of stack in CALL and RETURN instructions.
3. The first four instructions of a typical subroutine are :

```
PUSH PSW,      PUSH H  
PUSH B,        PUSH D
```

What will be the last five instructions of the subroutine? Explain clearly.

4. If the CALL and RET instructions are not provided in the 8085, could it be possible to write subroutines for this microprocessor ? If so how will you call and return from the subroutine ?
5. Discuss the passing parameters techniques in the subroutine.
6. Explain the necessity of subroutine documentation.
7. What do you mean by nested subroutines ?
8. Explain the re-entrant subroutine.
9. Explain the recursive subroutine with the help of example.

□□□

I/O and Memory Interface

6.1 Introduction

The important components of any computer system are, processor, memory, and I/O devices (peripherals). The processor fetches instructions (opcodes and operands/data) from memory, processes them and stores results in memory. The other components of the computer system (I/O devices) may be loosely called the Input/Output system. The main function of I/O system is to transfer information between processor or memory and the outside world.

6.1.1 Input / Output System

The Input/Output system has two major functions :

- Interface to the processor and memory via the system bus,
- Interface to one or more I/O devices by tailored data links

For clear understanding refer Fig. 6.1. Links to peripheral devices are used to exchange control, status and data between the I/O system and the external devices.

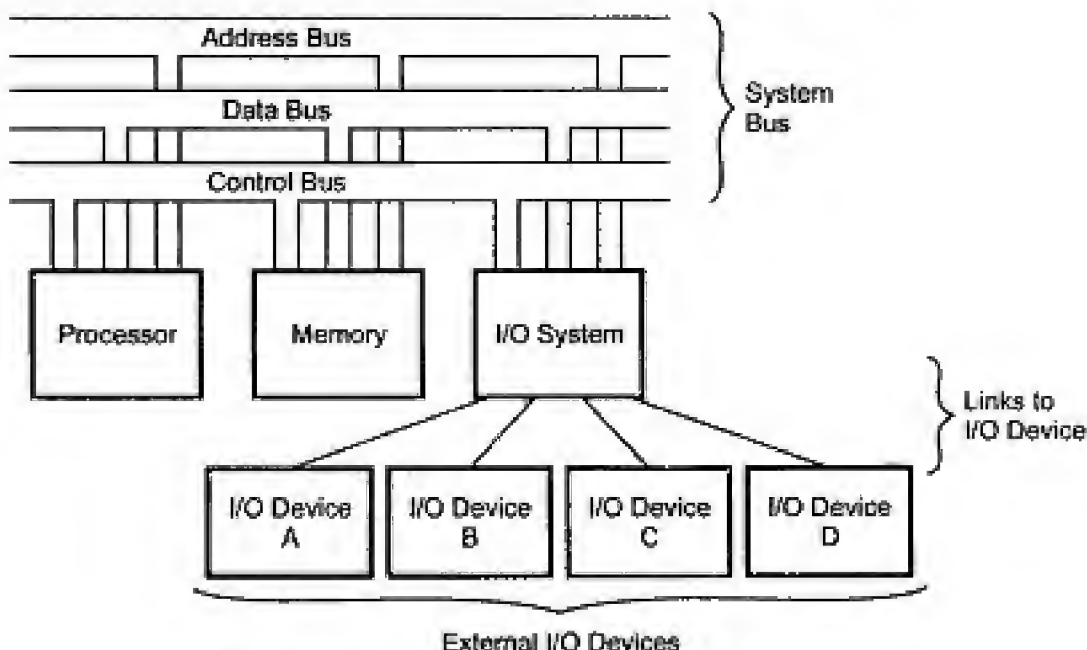


Fig. 6.1 Generic model of computer system

(6 - 1)

An important point that is to be noted here is, an I/O system is not simply mechanical connectors for connecting different devices required in the system to the system bus. It contains some logic for performing function of communication between the peripheral (I/O device) and the bus. The I/O system must have an interface internal to the computer (to the processor and memory) and an interface external to the computer (to the external I/O devices).

6.1.2 Requirements of I/O System

The I/O system is nothing but the hardware required to connect an I/O device to the bus. It is also called I/O interface. The major requirements of an I/O interface are :

1. Control and timing
2. Processor communication
3. Device communication
4. Data buffering
5. Error detection

All these requirements are explained here. The I/O interface includes a control and timing requirements to co-ordinate the flow of traffic between internal resources (memory, system bus) and external devices. Processor communication involves different types of signal transfers such as

- Processor sends commands to the I/O system which are generally the control signals on the control bus.
- Exchange of data between the processor and the I/O interface over the data bus.
- The data transfer rate of peripherals is often much slower than that of the processor. So it is necessary to check whether peripheral is ready or not for data transfer. If not, processor must wait. So it is important to know the status of I/O interface. The status signals such as BUSY, READY can be used for this purpose.
- A number of peripheral devices may be connected to the I/O interface. The I/O interface controls the communication of each peripheral with processor. So it must recognize one unique address for each peripheral connected to it.

The I/O interface must able to perform device communication which involves commands, status information and data.

Data buffering is also an essential task of an I/O interface. Data transfer rates of peripheral devices are quite high than that of processor and memory. The data coming from memory or processor are sent to an I/O interface, buffered and then sent to the peripheral device at its data rate. Also, data are buffered in I/O interface so as not to tie up the memory in a slow transfer operation. Thus the I/O interface must be able to operate at both peripheral and memory speeds.

I/O interface is also responsible for error detection and for reporting errors to the processor. The different types of errors are mechanical, electrical malfunctions reported by the device such as bad disk track, unintentional changes to the bit pattern, transmission errors etc. To fulfil all these requirements the important blocks necessary in any I/O interface are shown in Fig. 6.2.

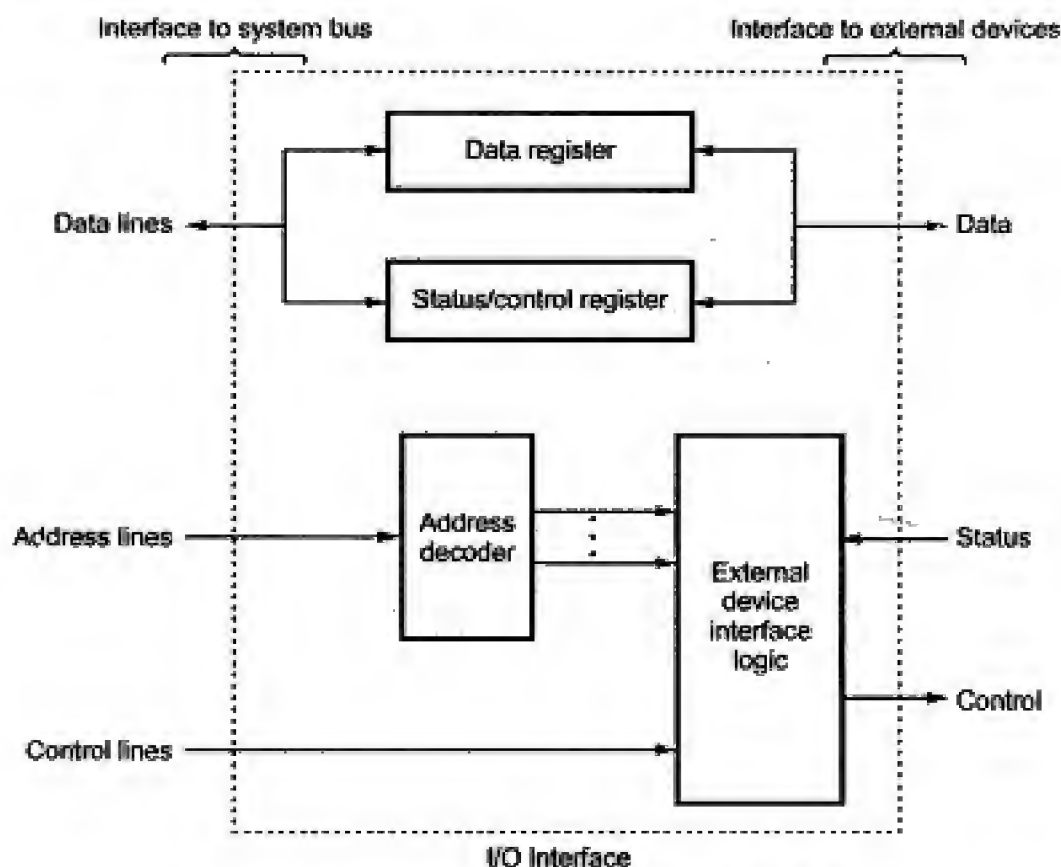
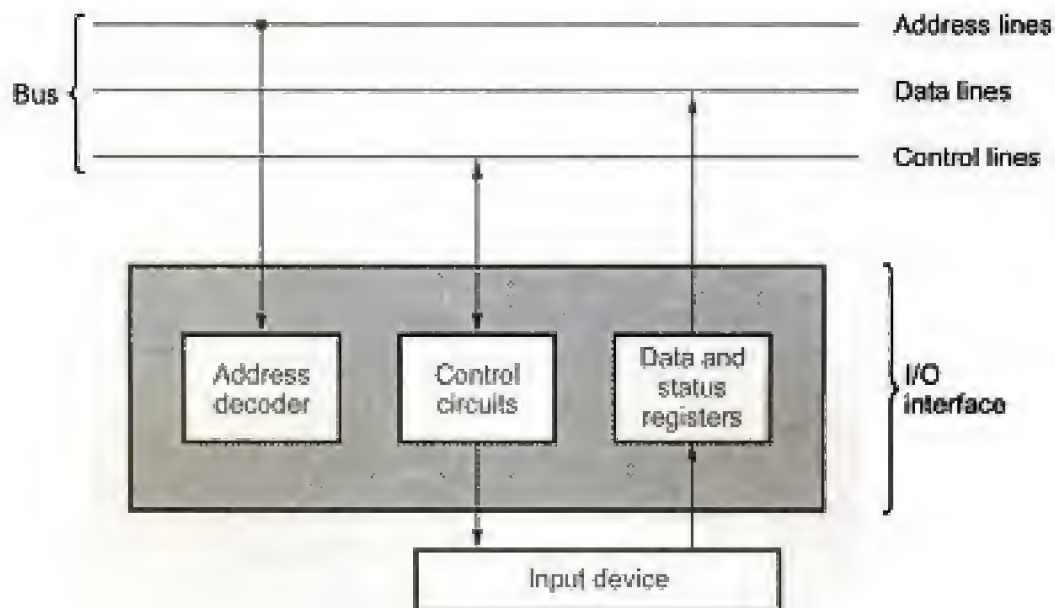


Fig. 6.2 Block diagram of I/O interface

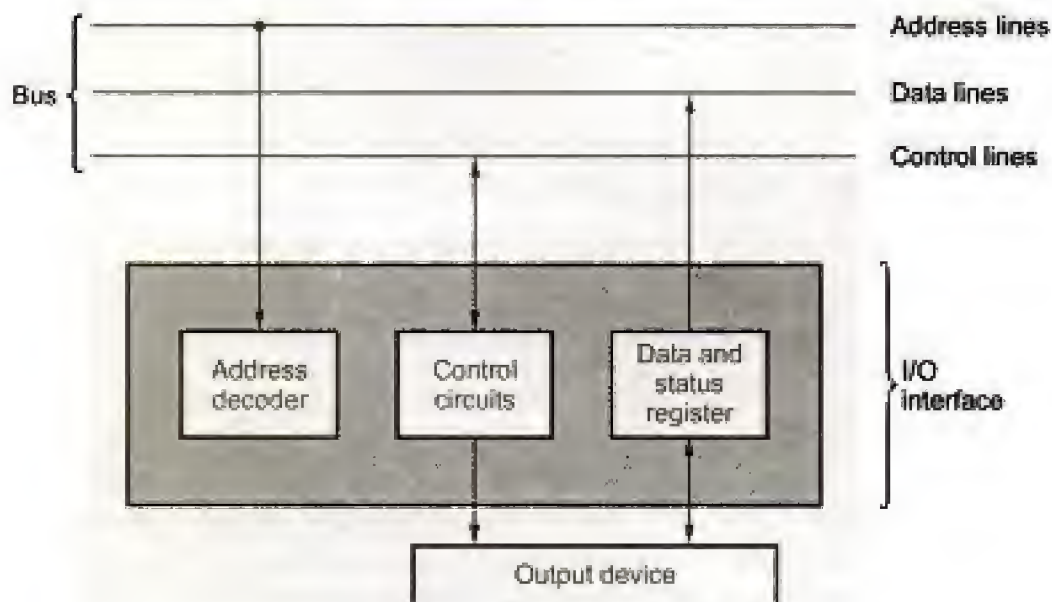
As shown in the Fig. 6.2, I/O interface consists of data register, status/control register, address decoder and external device interface logic. The data register holds the data being transferred to or from the processor. The status/control register contains information relevant to the operation of the I/O device. Both data and status/control registers are connected to the data bus. Address lines drive the address decoder. The address decoder enables the device to recognize its address when address appears on the address lines. The external device interface logic accepts inputs from address decoder, processor control lines and status signal from the I/O device and generates control signals to control the direction and speed of data transfer between processor and I/O devices.

The Fig. 6.3 shows the I/O interface for input device and output device. Here, for simplicity block schematic of I/O interface is shown instead of detail connections. The address decoder enables the device when its address appears on the address lines. The data register holds the data being transferred to or from the processor. The status register

contains information relevant to the operation of the I/O device. Both the data and status registers are assigned with unique addresses and they are connected to the data bus.



(a) I/O interface for input device

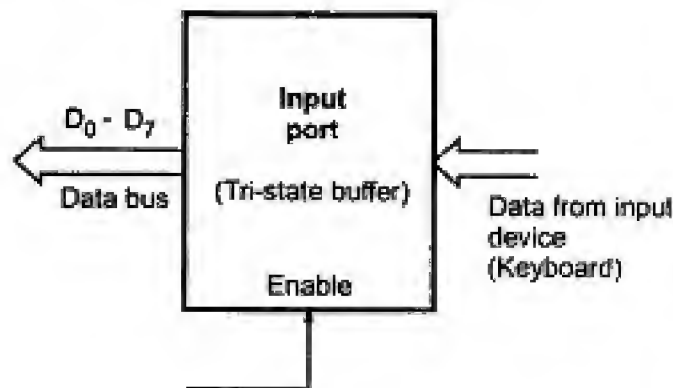


(b) I/O interface for output device

Fig. 6.3

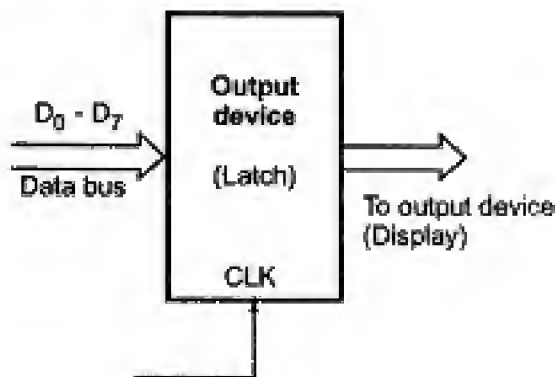
6.1.3 I/O Ports

The simplest form of I/O interface is an I/O port. The data transfer between microprocessor and input device is done with the help of input port. The data transfer between microprocessor and output device is done with the help of output port.

Input port :**Fig. 6.4**

It is used to read data from the input device such as keyboard. The simplest form of input port is a buffer. The input device is connected to the microprocessor through buffer as shown in the Fig. 6.4. This buffer is a tri-state buffer and its output is available only when enable signal is active. When microprocessor wants to read data from the input device (keyboard), the control signals from the microprocessor activates the buffer by asserting enable input of

the buffer. Once the buffer is enabled, data from the input device is available on the data bus. Microprocessor reads this data by initiating read command.

Output port :**Fig. 6.5**

It is used to send data to the output device such as display from the microprocessor. The simplest form of output port is a latch. The output device is connected to the microprocessor through latch, as shown in the Fig. 6.5. When microprocessor wants to send data to the output device, it puts the data on the data bus and activates the clock signal of the latch, latching the data from the data bus at the output of latch. It is then available at the output of latch for the output device.

6.2 I/O Data Transfer Techniques

In I/O data transfer, the system requires the transfer of data between I/O devices and microprocessor using I/O interface. It uses various techniques to perform I/O operations. These are :

- Program Controlled I/O or Programmed I/O or Polled I/O.
- Interrupt Driven I/O.
- Hardware Controlled I/O or DMA.

6.2.1 Programmed I/O

I/O operations will mean a data transfer between an I/O device and memory or between an I/O device and the processor. If in any computer system I/O operations are completely controlled by the processor, then that system is said to be using 'programmed I/O'. When such a technique is used, processor executes programs that initiate, direct and terminate the I/O operations, including sensing device status, sending a read or write command and transferring the data. It is the responsibility of the processor to periodically check the status of the I/O system until it finds that the operation is complete. Let us consider the following example.

The processor's software checks each of the I/O devices every so often. During this check, the microprocessor tests to see if any device needs servicing. Fig. 6.6 shows the flow chart for this. This is a simple program which services I/O ports A, B and C. The routine checks the status of I/O ports in proper sequence. It first transfers the status of I/O port A into the accumulator. Then the routine block checks the contents of accumulator to see if the service request bit is set. If it is, I/O port A service routine is called. After completion of service routine for I/O port A, the polling routine moves on to test port B and the process is repeated. This test and service procedure continues until all the I/O port status registers are tested and all the I/O ports requesting service are serviced. Once this is done, the processor continues to execute the normal programs.

The routine assigns priorities to the different I/O devices. Once the routine is started, the service request bit at port A is always checked first. Once port A is checked, port B is checked, and then port C. However, the order can be changed by simply changing the routine and thus the priorities.

When programmed I/O technique is used, processor fetches I/O related instructions from memory and **issues** I/O commands to I/O system to execute the instruction. The form of the **instruction** depends on the technique used for I/O addressing, i.e. memory mapped I/O or I/O mapped I/O.

As seen before, using memory mapped I/O technique, a memory reference instruction that causes data to be fetched from or stored at address specified automatically ~~becomes~~ an I/O instruction if that address is devoted to an I/O port. The usual memory load and store instructions are used to transfer a word of data to or from an I/O port.

When I/O mapped I/O technique is used, separate I/O instructions are required to activate READ I/O and WRITE I/O lines, which cause a word to be transferred between the addressed I/O port and the processor. The processor has two separate instructions, IN and OUT for I/O data transfer, e.g. the Intel 8085 microprocessor has two main I/O instructions. The **IN port address** instruction causes a word to be transferred from I/O port having address specified within the instruction to register A (accumulator) of 8085. The instruction **OUT port address** transfers a word from register A to I/O port having address specified within the instruction.

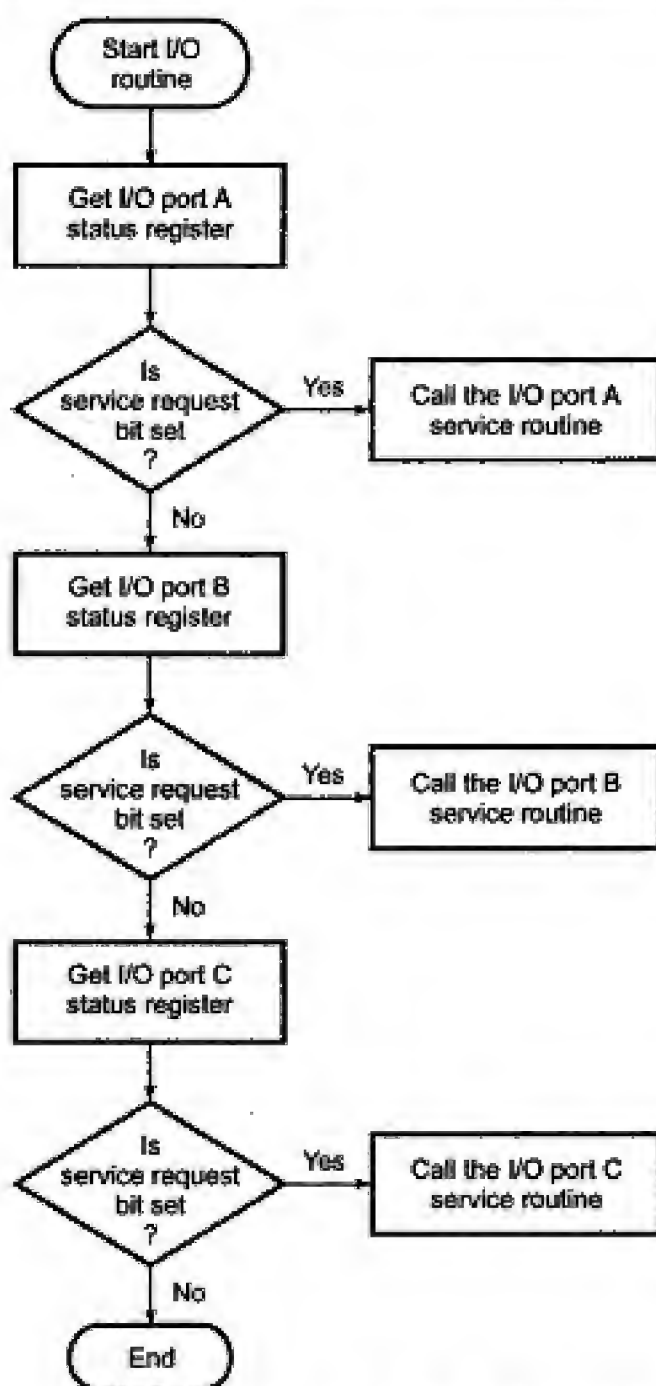


Fig. 6.6 Flowchart for I/O service routine

When an I/O instruction is encountered by the processor, the addressed I/O port is expected to be ready to respond to the instruction to prevent loss of information. Thus it is desirable for the processor to know the I/O device status (ready or not for I/O data transfer). In programmed I/O systems, the processor is usually programmed to test the I/O device status before initiating a data transfer.

6.2.2 Interrupt Driven I/O

Sometimes it is necessary to have the computer automatically execute one of a collection of special routines whenever certain conditions exists within a program or the computer system e.g. It is necessary that computer system should give response to devices such as keyboard, sensor and other components when they request for service.

The most common method of servicing such device is the **polled approach**. This is where the processor must test each device in sequence and in effect "ask" each one if it needs communication with the processor. It is easy to see that a large portion of the main program is looping through this continuous polling cycle. Such a method would have a serious and decremental effect on system throughput, thus limiting the tasks that could be assumed by the computer and reducing the cost effectiveness of using such devices.

A more desirable method would be the one that allows the processor to execute its main program and only stop to service peripheral devices when it is told to do so by the device itself. In effect, the method, would provide an external asynchronous input that would inform the processor that it should complete whatever instruction that is currently being executed and fetch a new routine that will service the requesting device. Once this servicing is completed, the processor would resume exactly where it left off. This method of servicing I/O request is called **Interrupt driven I/O**. It is easy to see that system throughput would drastically increase, and thus enhance its cost effectiveness. Most processors allow execution of special routines by interrupting normal program execution. When a processor is interrupted, it stops executing its current program and calls a special routine which "services" the interrupt. This is illustrated in Fig. 6.7. The event that causes the interruption is called **interrupt** and the special routine executed to service the interrupt is called **interrupt service routine(ISR)**.

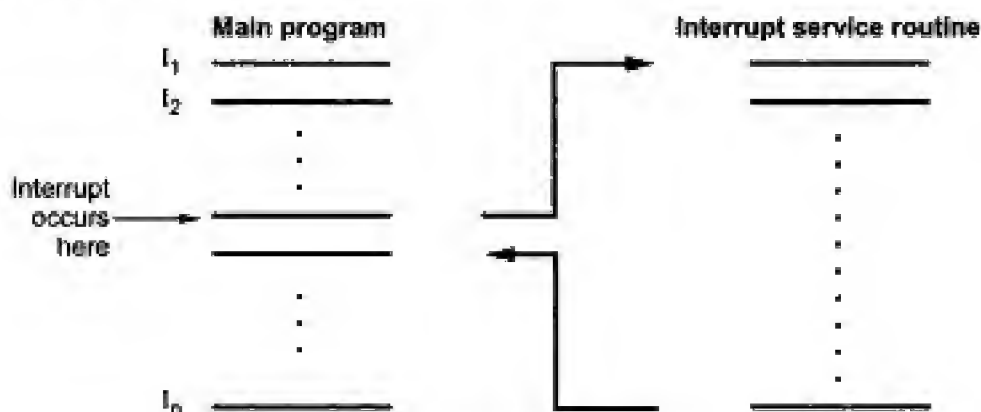


Fig. 6.7 Interrupt operation

6.2.3 Comparison between Programmed I/O and Interrupt Driven I/O

The Table 6.1 gives the comparison between programmed I/O and interrupt driven I/O.

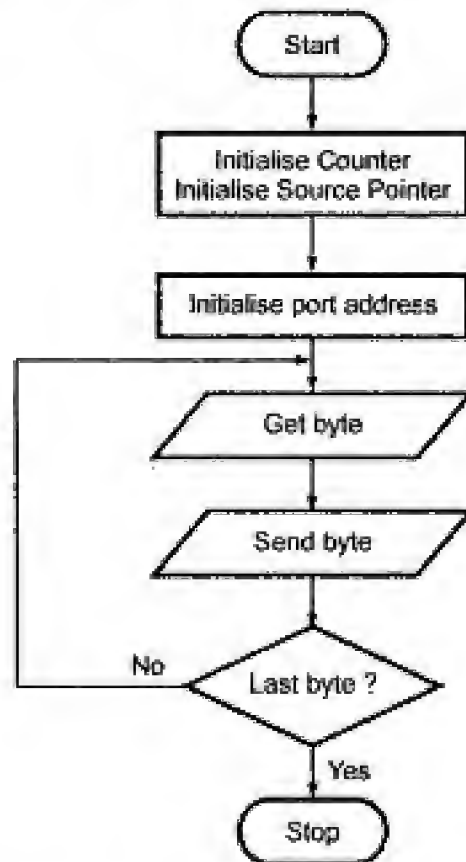
| Sr. No. | Programmed I/O | Interrupt Driven I/O |
|---------|--|--|
| 1. | In programmed I/O, processor has to check each I/O device in sequence and in effect 'ask' each one if it needs communication with the processor. This checking is achieved by continuous polling cycle and hence processor can not execute other instructions in sequence. | External asynchronous input is used to tell the processor that I/O device needs its service and hence processor does not have to check whether I/O device needs its service or not. |
| 2. | During polling processor is busy and therefore, have serious and decremental effect on system throughput. | In interrupt driven I/O, the processor is allowed to execute its instructions in sequence and only stop to service I/O device when it is told to do so by the device itself. This increases system throughput. |
| 3. | It is implemented without interrupt hardware support. | It is implemented using interrupt hardware support. |
| 4. | It does not depend on interrupt status. | Interrupt must be enabled to process interrupt driven I/O. |
| 5. | It does not need initialization of stack. | It needs initialization of stack. |
| 6. | System throughput decreases as number of I/O devices connected in the system increases. | System throughput does not depend on number of I/O devices connected in the system. |

Table 6.1 Comparison between programmed I/O and interrupt driven I/O

6.2.4 DMA Transfer

In software control data transfer, processor executes a series of instructions to carry out data transfer. For each instruction execution fetch, decode and execute phases are required. Fig. 6.8 gives the flowchart to transfer data from memory to I/O device.

Thus to carryout these tasks processor requires considerable time. So this method of data transfer is not suitable for large data transfers such as data transfer from magnetic disk or optical disk to memory. In such situations hardware controlled data transfer technique is used.

**Fig. 6.8**

There are two main drawbacks in both the techniques, programmed I/O and interrupt driven I/O :

1. The I/O transfer rate is limited by the speed with which the CPU can test and service a device.
2. The time that the CPU spends testing I/O device status and executing a number of instructions for I/O data transfers can often be better spent on other processing tasks.

To overcome above drawbacks an alternative technique, hardware controlled data transfer can be used.

6.2.4.1 Hardware Controlled Data Transfer

In this technique external device is used to control data transfer. External device generates address and control signals required to control data transfer and allows peripheral device to directly access memory. Hence this technique is referred to as direct memory access (DMA) and external device which controls the data transfer is referred to as DMA controller. Fig. 6.9 shows that how DMA controller operates in a computer system.

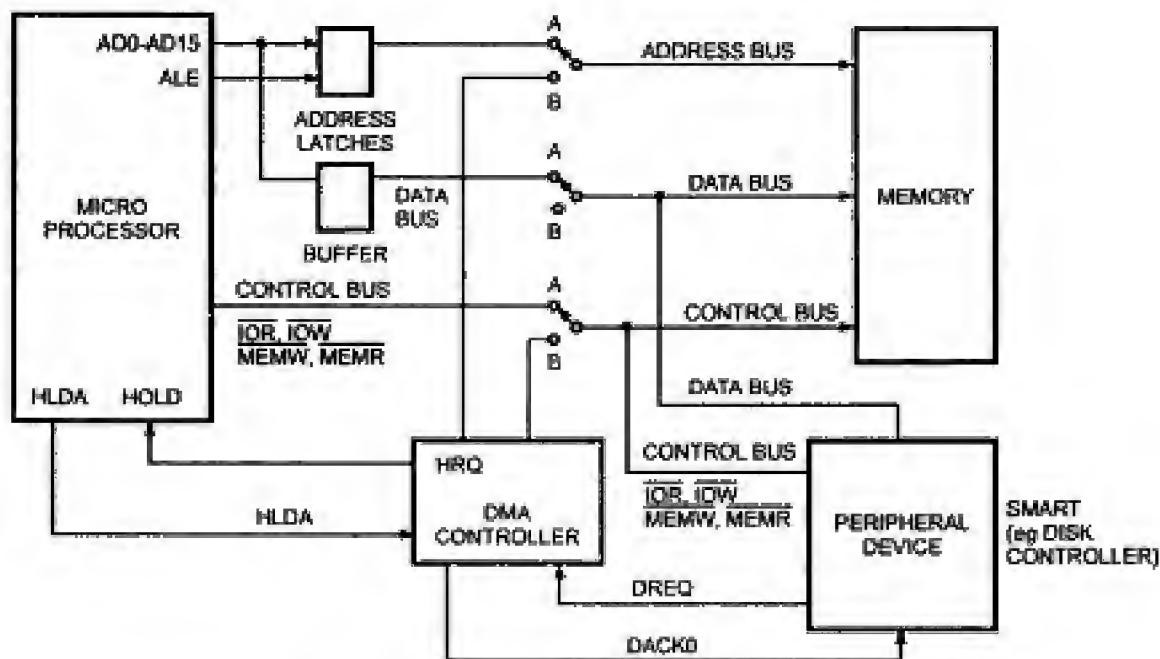


Fig. 6.9 DMA controller operating in a microprocessor system

6.2.4.2 DMA Idle Cycle

When the system is turned on, the switches are in the A position, so the buses are connected from the processor to the system memory and peripherals. Processor then executes the program until it needs to read a block of data from the disk. To read a block of data from the disk processor sends a series of commands to the disk controller device telling it to search and read the desired block of data from the disk. When disk controller is ready to transfer first byte of data from disk, it sends DMA request DRQ, signal to the DMA controller. Then DMA controller sends a hold request HRQ, signal to the processor HOLD input. The processor responds this HOLD signal by floating its buses and sending out a hold acknowledge signal HLDA, to the DMA controller. When the DMA controller receives the HLDA signal, it sends a control signal to change switch position from A to B. This disconnects the processor from the buses and connects DMA controller to the buses.

6.2.4.3 DMA Active Cycle

When DMA controller gets control of the buses, it sends the memory address where the first byte of data from the disk is to be written. It also sends a DMA acknowledge, DACK signal to the disk controller device telling it to get ready to output the byte. Finally, it asserts both the $\overline{\text{IOR}}$ and $\overline{\text{MEMW}}$ signals on the control bus. Asserting the $\overline{\text{IOR}}$ signal enables the disk controller to output the byte of data from the disk on the data bus and asserting the $\overline{\text{MEMW}}$ signal enables the addressed memory to accept data from the data bus. In this technique data is transferred directly from the disk controller to the memory location without passing through the processor or the DMA controller.

Thus, the CPU is involved only at the beginning and end of the transfer. The data transfer is monitored by DMA controller, which is also called DMA channel. When the CPU wishes to read or write a block of data, it issues a command to the DMA module or DMA channel by sending the following information to the DMA channel/controller :

- 1) A read or write operation.
- 2) The address of I/O device involved.
- 3) The starting address in memory to read from or write to.
- 4) The number of words to be read or written.

6.2.5 Data Transfer Modes

DMA controller transfer data in one of the following three modes :

- Single transfer mode (cycle stealing)
- Block transfer mode
- Demand or burst transfer mode

Single Transfer Mode

In this mode device can make only one transfer (byte or word). After each transfer DMAC gives the control of all buses to the processor. Due to this processor can have access to the buses on a regular basis.

It allows the DMAC to time share the buses with the processor, hence this mode is most commonly used.

The operation of the DMA in a single transfer mode is as given below :

1. I/O device asserts DRQ line when it is ready to transfer data.
2. The DMAC asserts HLDA line to request use of the buses from the processor.
3. The processor asserts HLDA, granting the control of buses to the DMAC.
4. The DMAC asserts \overline{DACK} to the requesting I/O device and executes DMA bus cycle, resulting data transfer.
5. I/O device deasserts its DRQ after data transfer of one byte or word.
6. DMA deasserts \overline{DACK} line.
7. The word/byte transfer count is decremented and the memory address is incremented.
8. The HOLD line is deasserted to give control of all buses back to the processor.
9. HOLD signal is reasserted to request the use of buses when I/O device is ready to transfer another byte or word. The same process is then repeated until the last transfer.
10. When the transfer count is exhausted, terminal count is generated to indicate the end of the transfer.

Block Transfer Mode

In this mode device can make number of transfers as programmed in the word count register. After each transfer word count is decremented by 1 and the address is decremented or incremented by 1. The DMA transfer is continued until the word count "rolls over" from zero to FFFFH, a Terminal Count (TC) or an external END of Process (EOP) is encountered. Block transfer mode is used when the DMAC needs to transfer a block of data.

The operation of DMA in block transfer mode is as given below :

1. I/O device asserts DRQ line when it is ready to transfer data.
2. The DMAC asserts HLDA line to request use of the buses from the microprocessor.
3. The microprocessor asserts HLDA, granting the control of buses to the DMAC.
4. The DMAC asserts \overline{DACK} to the requesting I/O device and executes DMA bus cycle, resulting data transfer.
5. I/O device deasserts its DRQ after data transfer of one byte or word.
6. DMA deasserts \overline{DACK} line.
7. The word/byte transfer count is decremented and the memory address is incremented.
8. When the transfer count is exhausted, the data transfer is not complete and the DMAC waits for another DMA request from the I/O device, indicating that it has another byte or word to transfer. When DMAC receives DMA request steps through are repeated.
9. If the transfer count is not exhausted, the data transfer is complete then DMAC deasserts the HOLD to tell the microprocessor that it no longer needs the buses.
10. Microprocessor then deasserts the HLDA signal to tell the DMAC that it has resumed control of the buses.

Demand Transfer Mode

In this mode the device is programmed to continue making transfers until a TC or external \overline{EOP} is encountered or until DREQ goes inactive.

The operation of DMA in demand transfer mode is as given below :

1. I/O device asserts DRQ line when it is ready to transfer data.
2. The DMAC asserts HLDA line to request use of the buses from the microprocessor.
3. The microprocessor asserts HLDA, granting the control of buses to the DMAC.
4. The DMAC asserts \overline{DACK} to the requesting I/O device and executes DMA bus cycle, resulting data transfer.
5. I/O device deasserts its DRQ after data transfer of one byte or word.
6. DMA deasserts \overline{DACK} line.

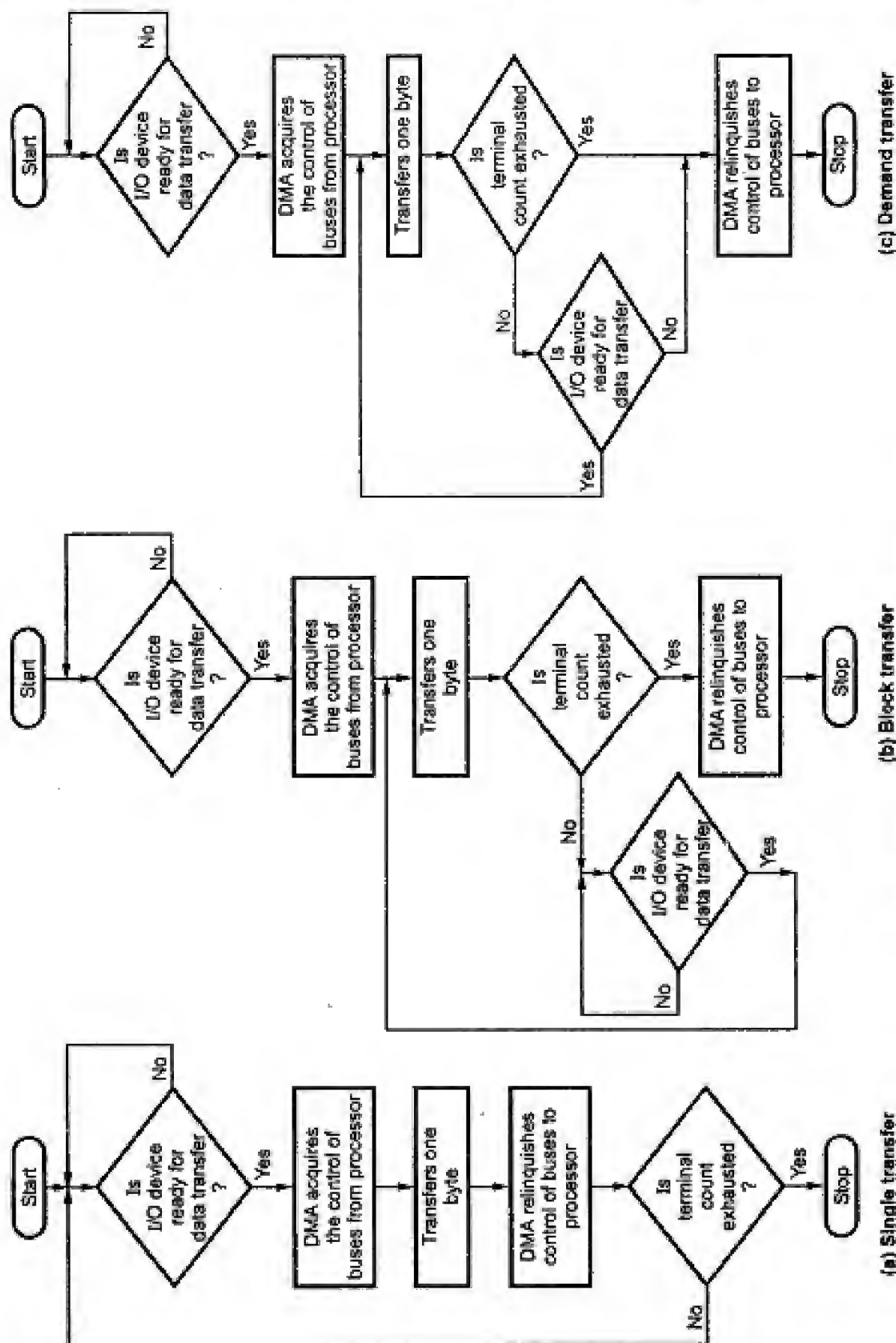


Fig. 6.10 Three data transfer modes of DMA

7. The word/byte transfer count is decremented and the memory address is incremented.
8. The DMAC continues to execute transfer cycles until the I/O device deasserts DRQ indicating its inability to continue delivering data. The DMAC deasserts HOLD signal, giving the buses back to microprocessor. It also deasserts \overline{DACK} .
9. I/O device can re-initiate demand transfer by reasserting DRQ signal.
10. Transfer continues in this way until the transfer count has been exhausted.

The flowcharts in the Fig. 6.10 summarized the three data transfer modes of DMA.

6.3 I/O Interfacing Techniques in 8085

The most of the microprocessors support isolated I/O system. It partitions memory from I/O, via software, by having instructions that specifically access (address) memory, and others that specifically access I/O. When these instructions are decoded by the microprocessor, an appropriate control signal is generated to activate either memory or I/O operation. In 8085, $\overline{IO/\overline{M}}$ signal is used for this purpose. The 8085 outputs a logic '1' on the $\overline{IO/\overline{M}}$ line for an I/O operation and a logic '0' for memory operation. In 8085, it is possible to connect 64 Kbyte memory and 256 I/O ports in the system since 8085 sends 16 bit address for memory and 8 bit address for I/O. I/O devices can be interfaced to an 8085A system in two ways :

1. I/O Mapped I/O
2. Memory mapped I/O

6.3.1 I/O Mapped I/O

In I/O mapped I/O, the 8085 uses $\overline{IO/\overline{M}}$ signal to distinguish between I/O read/write and memory read/write operations. The 8085 has separate instructions IN and OUT for I/O data transfer. When 8085 executes IN or OUT instruction, it places device address (port number) on the demultiplexed low order address bus as well as the high order address bus. In other words, we can say that higher order address bus duplicates the contents of demultiplexed low-order address bus, when 8085 microprocessor executes an IN or OUT instruction. For example, if the device address is 60H then the contents on A_{15} to A_0 will be as follows :

| A_{15} | A_{14} | A_{13} | A_{12} | A_{11} | A_{10} | A_9 | A_8 | A_7 | A_6 | A_5 | A_4 | A_3 | A_2 | A_1 | A_0 |
|----------|----------|----------|----------|----------|----------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |

Here, A_8 follows A_0 , A_9 follows A_1 and so on, as shown below.

| A_7 | A_6 | A_5 | A_4 | A_3 | A_2 | A_1 | A_0 | Device Address |
|----------|----------|----------|----------|----------|----------|-------|-------|----------------|
| A_{15} | A_{14} | A_{13} | A_{12} | A_{11} | A_{10} | A_9 | A_8 | |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | |
| | | | | | | | | 60H |

The instruction IN inputs data from an input device (such as keyboard) into the accumulator and the instruction OUT sends the contents of the accumulator to an output device such as LED display. These are two byte instructions. The second byte of the instruction specifies the address or the port number of an I/O device. As it is a byte, the address or port number can be any of the 256 combinations of eight bits, from 00H to FFH. Therefore, the 8085 can communicate with 256 different I/O devices. When we want to interface an I/O device, it is necessary to assign a device address or a port number.

6.3.2 I/O Device Selection

As mentioned earlier, the 8085 gives 8 bit I/O address. This means it can select one of the 256 I/O ports. To select an appropriate I/O device, it is necessary to do following things.

1. Decode the address to generate unique signal corresponding to the device address on the bus.
2. When device address signal and control signal ($\overline{\text{IOR}}$ or $\overline{\text{IOW}}$) both are low, generate device select signal.
3. Use device select signal to activate the interfacing device (I/O port).

Fig. 6.11 shows the absolute decoding circuit for the I/O device. The IC 74LS138, 3:8 decoder along with 3 input OR gate is used to generate device select signal. This signal is ORed with control signal ($\overline{\text{IOR}}$ or $\overline{\text{IOW}}$) to generate device enable signal.

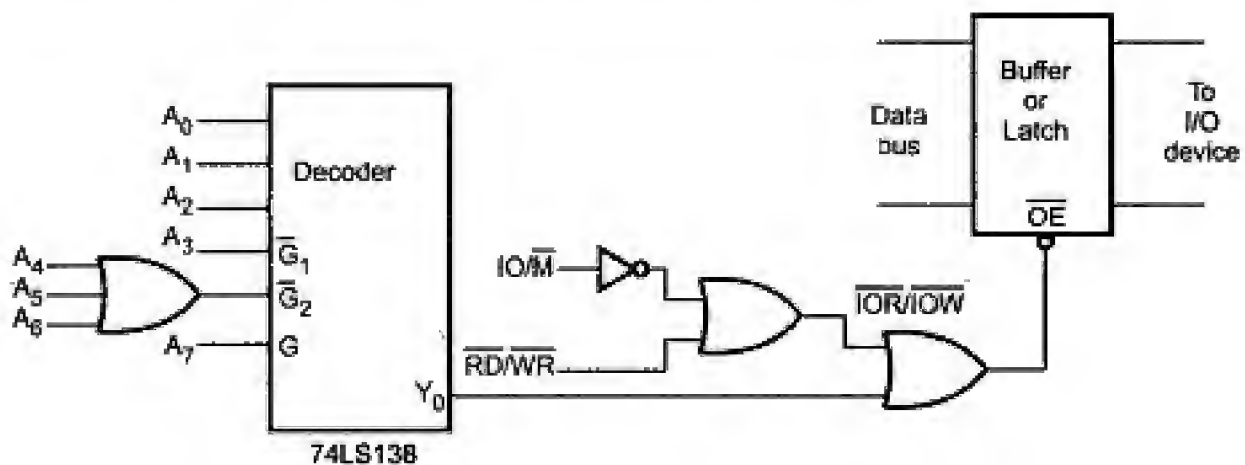


Fig 6.11 Absolute decoding circuit for the I/O device

To generate device select signal (Y_0) low, the address on the address bus must be as given below :

$A_2 A_1 A_0 = 000\text{H}$; Activates Y_0 output

$A_3 A_4 A_5 A_6 = 0000\text{H}$; Makes \overline{G}_1 and \overline{G}_2 low to enable output of decoder

$A_7 = 1\text{H}$; Makes G high to enable output of decoder

Note : Decoder output is enabled only when control signals \overline{G}_1 and \overline{G}_2 are low and control signal G is high. Therefore the address of this I/O device is 80H as shown in the table 6.2.

| A ₇ | A ₆ | A ₅ | A ₄ | A ₃ | A ₂ | A ₁ | A ₀ | Address |
|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|---------|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 80H |

Table 6.2

6.3.3 Interfacing Input and Output Devices with Examples

Interfacing Input Device :

The microprocessor 8085 accepts 8 bit data from the input device such as keyboard, sensors, transducers etc. Fig. 6.12 shows the circuit diagram to interface input port (buffer) which is used to read the status of 8 switches. The address for this input device is 80H as device select signal goes low when address is 80H.

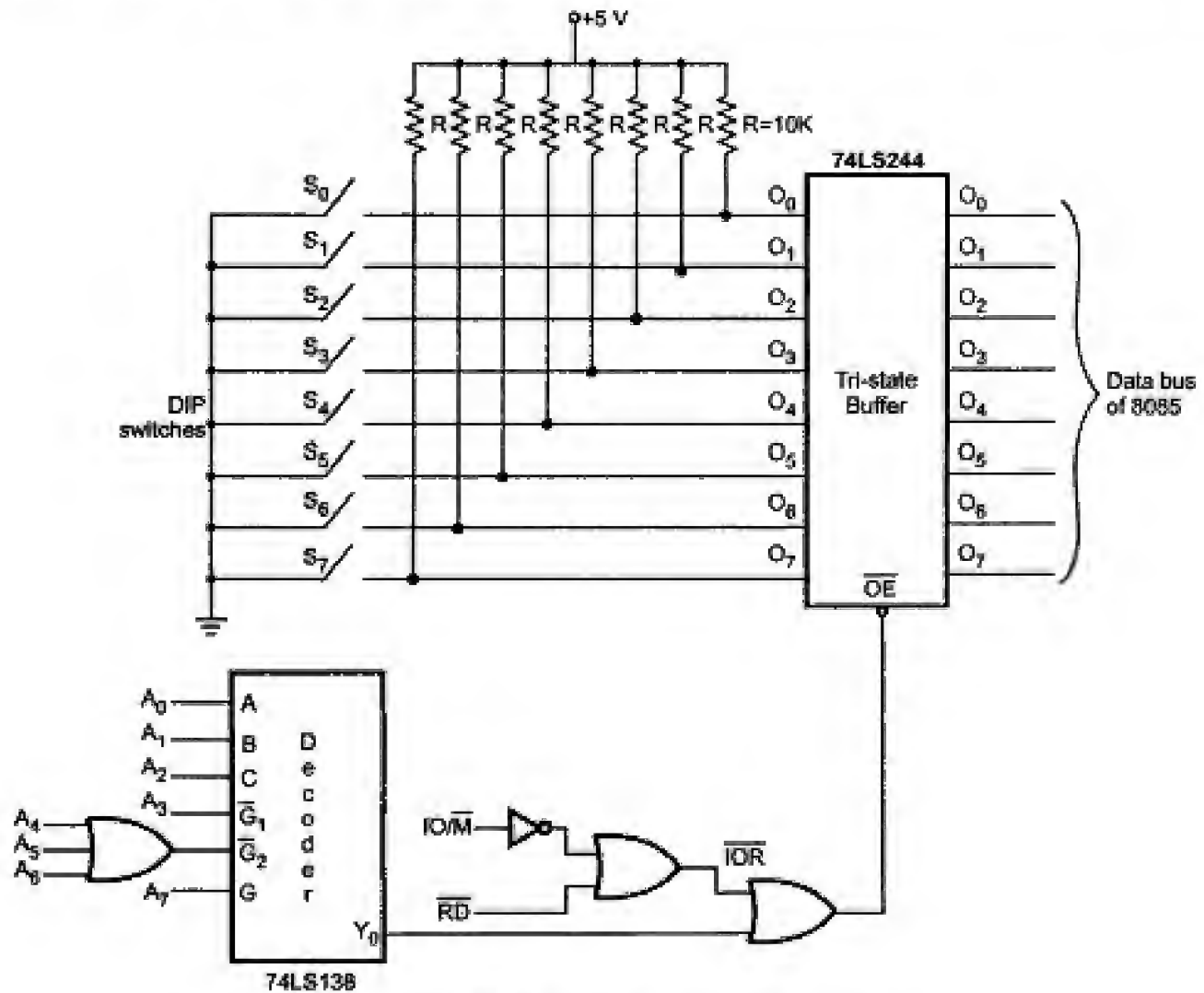


Fig. 6.12 Circuit diagram to Interface input port

When the switch is in the released position, the status of line is high otherwise status is low. With this information microprocessor can check a particular key is pressed or not.

The following program checks whether the switch 2 is pressed or not.

Program :

```
IN 80H      ; Read status of all switches
ANI 02H     ; Mask bit positions for other switches
JZ NEXT     ; if program control is transferred to label
            ; NEXT, then switch 2 is pressed otherwise not.
```

Interfacing Output Device :

The microprocessor 8085 sends 8 bit data to the output device such as 7 segment displays, LEDs, printer etc. Fig 6.13 shows the circuit diagram to interface output port (latch) which is used to send the signal for glowing the LEDs. LED will glow when output pin status is low. The IC 74LS138 and 3 input OR gate is used to generate device select signal. The latch enable signal is active high. So NOR gate is used to generate latch enable signal, which goes high when Y_1 and \overline{IOW} both are low.

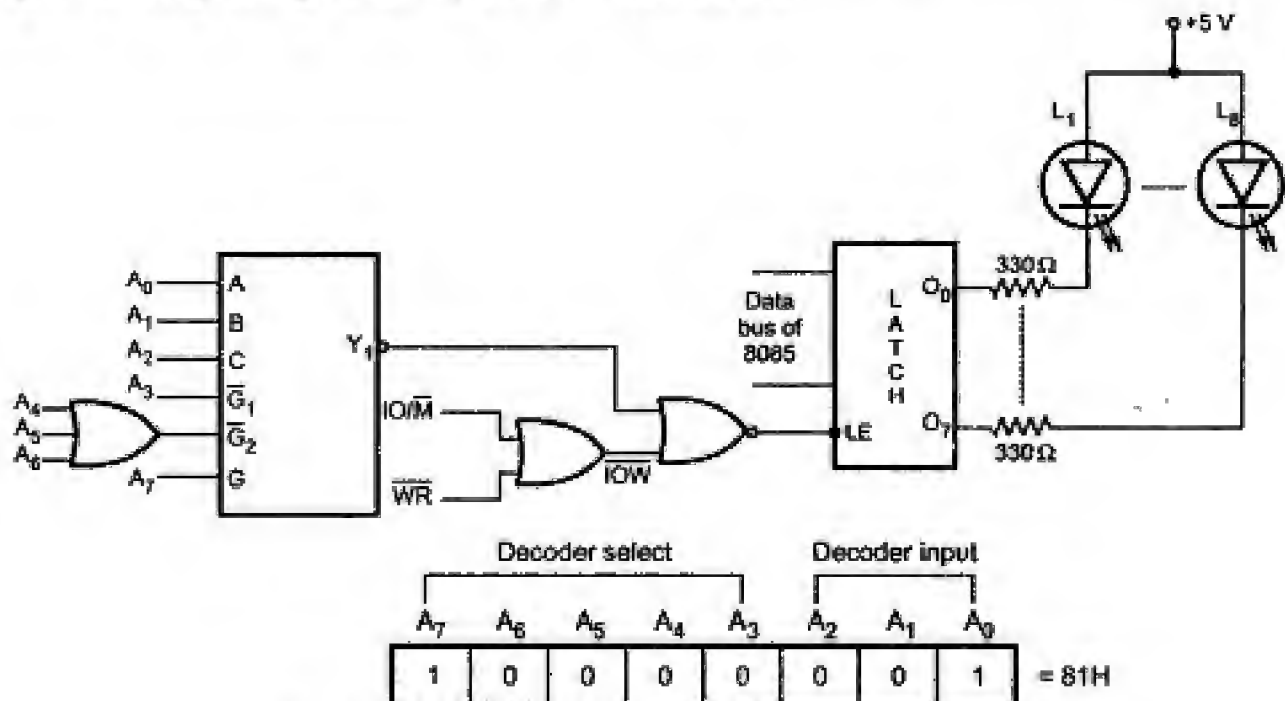


Fig. 6.13 Circuit diagram to interface output port

The following program glows the LEDs L_1 , L_3 and L_6 .

| L ₈ | L ₇ | L ₆ | L ₅ | L ₄ | L ₃ | L ₂ | L ₁ | |
|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|-------|
| 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | = DAH |

The code (data) DAH must be sent on the latch to glow LEDs L_1 , L_3 and L_6 .

Program :

```
MVI A, DAH ; Loads the data in the accumulator.
OUT 81H    ; sends the data on the latch.
```

The Fig. 6.14 shows the combined circuit for I/O interfacing. For this circuit the address of input port is 80H and address of output port is 81H. The following program displays the status of switches on the LEDs.

Program :

```
IN 80H           ; Read status of all switches.
OUT 81H          ; send status on the output port.
```

➔ **Example 6.1 :** Refer Fig. 6.14 and write a program that will check the switch1 status and do accordingly.

1. SW1 = 0 : Blink lower nibble LEDs.
2. SW1 = 1 : Blink Higher nibble LEDs.

Assume delay routine is available.

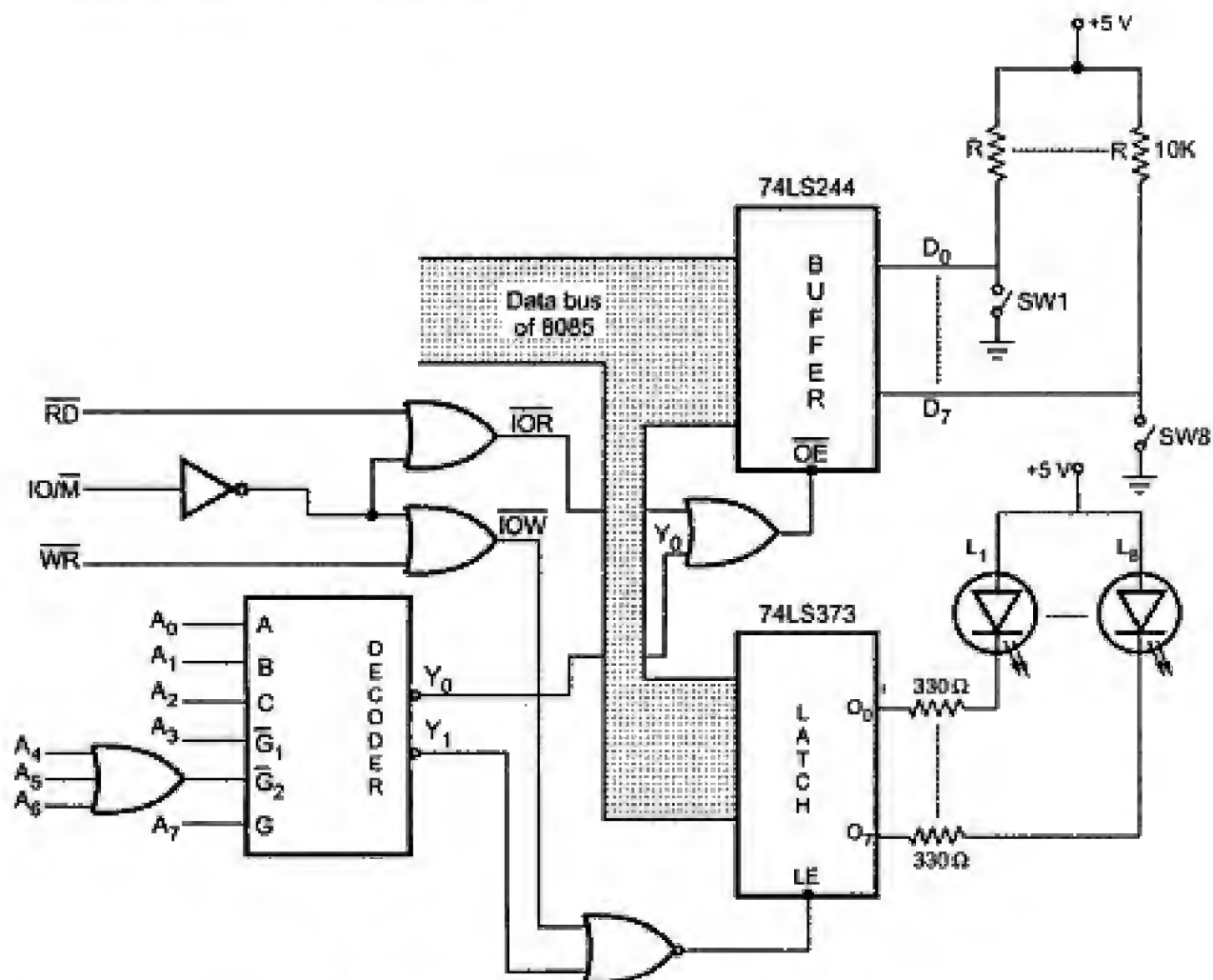


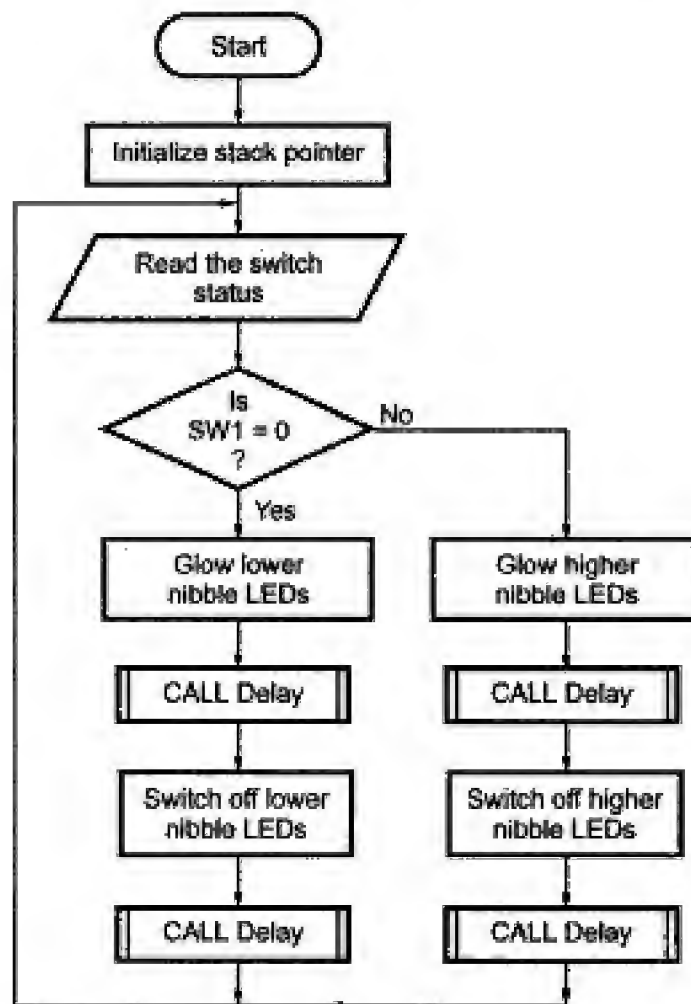
Fig. 6.14 I/O Interfacing using I/O mapped I/O

Solution :

Input port address = 80H

Output port address = 81H

Flowchart :



Source Program :

```

      LXI SP, 27FFH      ; Initialize stack pointer
START : IN 80H           ; Read status of switches
      ANI 01H           ; Masks Bit 1 to Bit 7
      JNZ HIGHER        ; If sw1 status is not zero goto blink
                           ; higher nibble
      MVI A, F0H        ; Load bit pattern to glow lower nibble
                           ; LEDs
      OUT 81H           ; Send it to output port
      CALL Delay         ; Call delay subroutine
      MVI A, FFH        ; Load bit pattern to switch off all LEDs
      OUT 81H           ; Send it to output port
      CALL Delay         ; Call delay subroutine
      JMP START         ; JUMP to START
  
```

```
HIGHER : MVI A, 0FH ; Load bit pattern to glow higher nibble LEDs
          OUT 81H    ; Send it to output port
          CALL Delay ; Call delay subroutine
          MVI A, FFH ; Load bit pattern to switch off all LEDs
          OUT 81H    ; Send it to output port
          CALL Delay ; Call delay subroutine
          JMP START  ; JUMP to START
```

6.3.4 Memory Mapped I/O

In memory mapped I/O, the I/O devices are assigned and identified by 16 bit addresses. The memory related instructions transfer the data between an I/O device and the microprocessor, as long as I/O port is assigned to the memory address space rather than to the I/O address space. The register associated with the I/O port is simply treated as a memory location. Thus I/O device becomes a part of the system's memory map and hence its name. In memory-mapped I/O every instruction that refers to a memory location can control I/O. The source and destination of the data is limited with I/O mapped I/O, since for an IN instruction the destination register is always the accumulator, and for the OUT instruction the source register is always the accumulator. However, for memory mapped I/O there are number of sources and destinations.

| Instructions | Interpretation for memory mapped I/O |
|--------------|---|
| MOV r, M | ; Input from a port to specified register. |
| LDA addr | ; Input from a port to accumulator. |
| LHLD addr | ; Input from two ports to HL register pair. |
| ADD M | ; Port contents are added into accumulator ; contents and result is stored in the accumulator. |
| ANA M | ; Port contents are logically ANDed with the accumulator ; contents and result is stored in the accumulator. |
| ORA M | ; Port contents are logically ORed with the accumulator ; contents and result is stored in the accumulator. |
| XRA M | ; Port contents are logically XORed with the accumulator ; contents and result is stored in the accumulator. |
| CMP M | ; Compares the port contents with the accumulator ; contents and updates the flag register contents accordingly. |
| MOV M,r | ; Outputs specified register contents to the port. |
| STA addr | ; Outputs accumulator contents to the port. |
| SHLD addr | ; Outputs HL register contents to two ports. |
| MVI M, data | ; Outputs immediate data to the port. |

Interfacing of I/O port with memory mapped I/O

In memory mapped I/O, $\overline{\text{MEMR}}$ (memory read) and $\overline{\text{MEMW}}$ (memory write) control signals are required to control the data transfer between I/O device and microprocessor. As 8085 gives 16 bit memory address, it is necessary to decode 16 bit memory address to generate device select signal in case of memory mapped I/O. Fig.6.15 shows the Interfacing of I/O devices in memory mapped I/O mode.

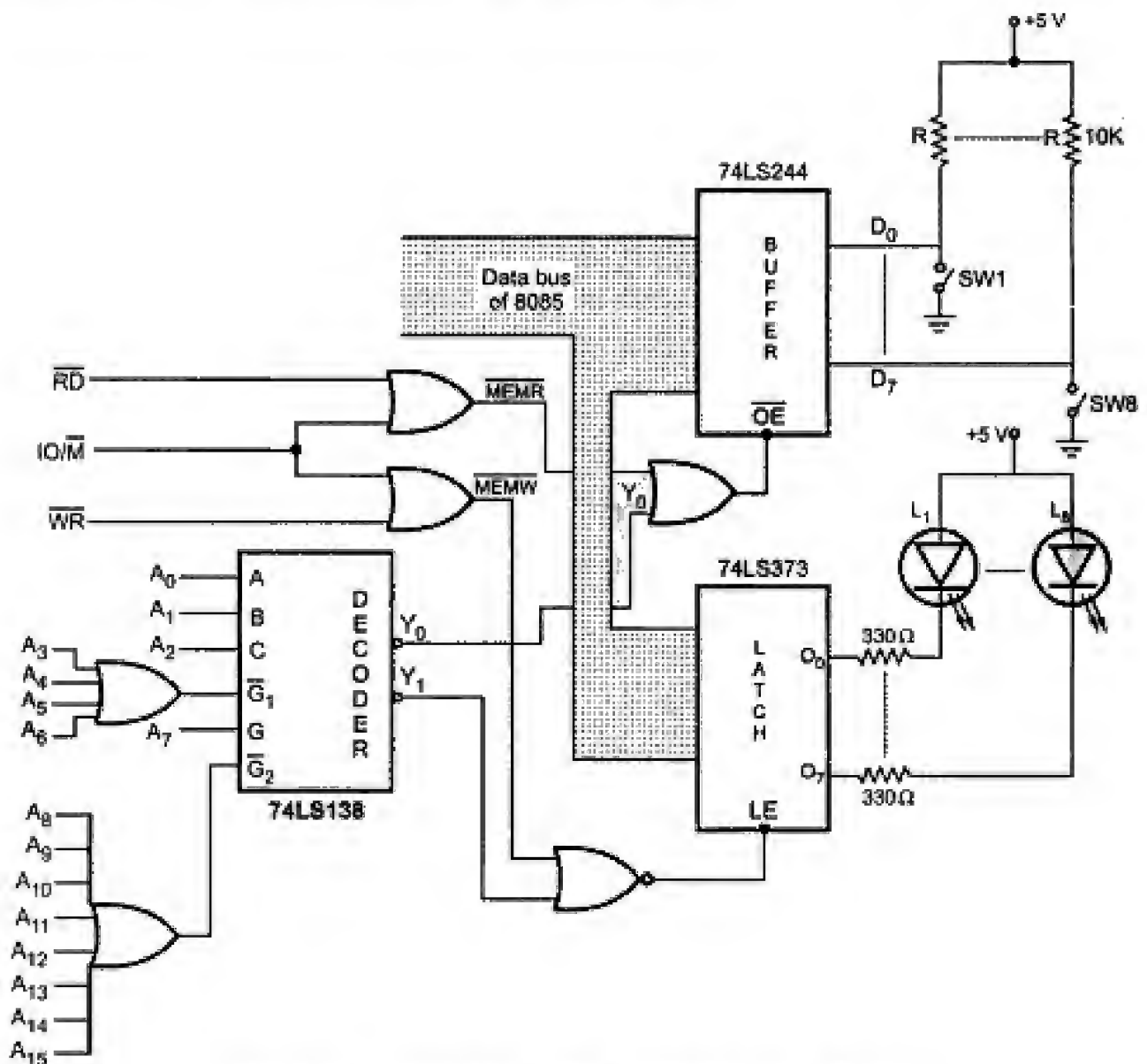


Fig. 6.15 I/O interfacing using memory mapped I/O

➡ **Example 6.2 :** Identify the port address and the mapping scheme for the Fig. 6.16 given below.

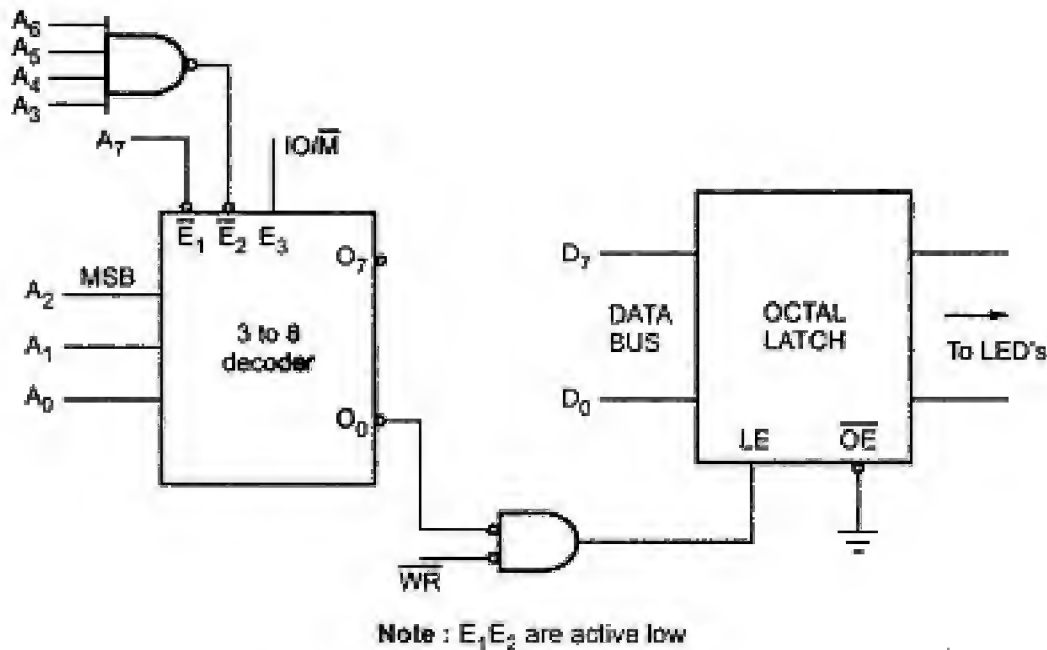


Fig. 6.16

Solution : Mapping scheme : I/O mapped I/O

| Address lines | | | | | | | | Port Address |
|---------------|-------|-------|-------|-------|-------|-------|-------|--------------|
| A_7 | A_6 | A_5 | A_4 | A_3 | A_2 | A_1 | A_0 | |
| 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 78H |

6.3.5 Comparison between I/O Mapped I/O and Memory Mapped I/O

| Sr. No. | Memory mapped I/O | I/O mapped I/O |
|---------|--|--|
| 1. | In this device address is 16 bit. Thus A_0 to A_{15} lines are used to generate device address. | In this I/O device address is 8 bit. Thus A_0 to A_7 or A_8 to A_{15} lines are used to generate device address. |
| 2. | $\overline{\text{MEMR}}$ and $\overline{\text{MEMW}}$ control signals are used to control read and write I/O operations. | $\overline{\text{IOR}}$ and $\overline{\text{IOW}}$ control signals are used to control read and write I/O operations. |
| 3. | Instructions available are LDA addr, STA addr, LDAX rp, STAX rp, MOV M,R, MOV R,M, ADD M, CMP M etc. Hence we can use direct as well as indirect addressing modes. | Instructions available are IN and OUT. Only direct addressing can be used. |

| | | |
|----|---|--|
| 4. | Data transfer is between any register and I/O device. | Data transfer is between accumulator and I/O device. |
| 5. | Maximum number of I/O devices are 65536 (theoretically). | Maximum number of I/O devices are 256. |
| 6. | Execution speed using LDA addr, STA addr is 13 T-state and 7 T-states for MOV M, r and MOV r, M instructions. | Execution speed is 10 T-states. |
| 7. | Decoding 16 bit address may require more hardware. | Decoding 8 bit address will require less hardware. |

6.4 Memory Interfacing

Memory is an integral part of a microprocessor system, and in this section, we will discuss how to interface a memory device with the microprocessor. The memory interfacing circuit is used to access memory quite frequently to read instruction codes and data stored in memory. This read/write operations are monitored by control signals. The microprocessor activates these signals when it wants to read from and write into memory.

6.4.1 Basic Concepts in Memory Interfacing

For interfacing memory devices to microprocessor 8085, following important points are to be kept in mind.

1. Microprocessor 8085 can access 64 Kbytes memory since address bus is 16-bit. But it is not always necessary to use full 64 Kbytes address space. The total memory size depends upon the application.
2. Generally EPROM (or EPROMs) is used as a program memory and RAM (or RAMs) as a data memory. When both, EPROM and RAM are used, the total address space 64 Kbytes is shared by them.
3. The capacity of program memory and data memory depends on the application.
4. It is not always necessary to select 1 EPROM and 1 RAM. We can have multiple EPROMs and multiple RAMs as per the requirement of application.

For example :

We have to implement 32 Kbytes of program memory and 4 Kbytes EPROMs are available. In this case, we can connect 8 EPROMs in parallel ($4 \text{ Kbytes} \times 8 = 32 \text{ Kbytes}$) with different chip select for each EPROM.

5. We can place EPROM/RAM anywhere in full 64 Kbytes address space. But program memory (EPROM) should be located from address 0000H since reset address of 8085 microprocessor is 0000H.

6. It is not always necessary to locate EPROM and RAM in consecutive memory addresses. For example : If the mapping of EPROM is from 0000H to 0FFFH, it is not must to locate RAM from 1000H. We can locate it anywhere between 1000H and FFFFH. Where to locate memory component totally depends on the application.

The memory interfacing requires to :

- Select the chip
- Identify the register
- Enable the appropriate buffer.

Microprocessor system includes memory devices and I/O devices. It is important to note that microprocessor can communicate (read/write) with only one device at a time, since the data, address and control buses are common for all the devices. In order to communicate with memory or I/O devices, it is necessary to decode the address from the microprocessor. Due to this each device (memory or I/O) can be accessed independently. The following section describes common address decoding techniques.

Address Decoding Techniques :

- Absolute decoding/Full Decoding
- Linear decoding/Partial Decoding

Absolute decoding

In absolute decoding technique, all the higher address lines are decoded to select the memory chip, and the memory chip is selected only for the specified logic levels on these high-order address lines; no other logic levels can select the chip. Fig. 6.17 shows the memory interface with absolute decoding. This addressing technique is normally used in large memory systems.

Memory Map :

| Memory ICs | A ₁₅ | A ₁₄ | A ₁₃ | A ₁₂ | A ₁₁ | A ₁₀ | A ₉ | A ₈ | A ₇ | A ₆ | A ₅ | A ₄ | A ₃ | A ₂ | A ₁ | A ₀ | Address |
|---------------------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|---------|
| Starting address of EPROM | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0000H |
| End address of EPROM | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 03FFFH |
| Starting address of RAM | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2000H |
| End address of RAM | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 23FFFH |

Table 6.3

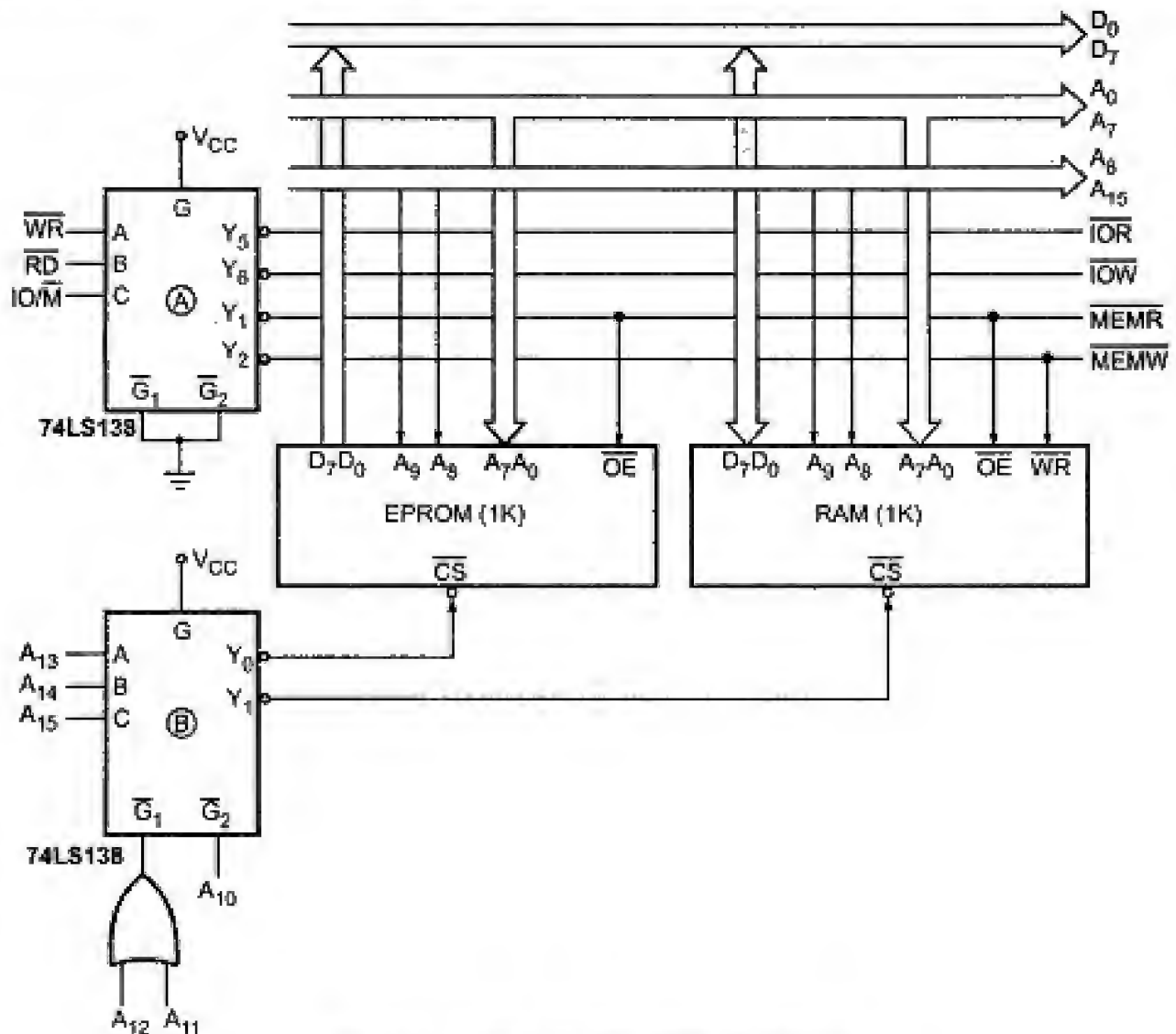


Fig. 6.17 Absolute decoding technique

Linear decoding

In small systems, hardware for the decoding logic can be eliminated by using individual high-order address lines to select memory chips. This is referred to as linear decoding. Fig. 6.18 shows the addressing of RAM with linear decoding technique. This technique is also called **partial decoding**. It reduces the cost of decoding circuit, but it has a drawback of multiple addresses (shadow addresses).

Fig. 6.18 shows the addressing of RAM with linear decoding technique. A₁₅ address line, is directly connected to the chip select signal of EPROM and after inversion it is connected to the chip select signal of the RAM. Therefore, when the status of A₁₅ line is 'zero', EPROM gets selected and when the status of A₁₅ line is 'one' RAM gets selected. The status of the other address lines is not considered, since those address lines are not used for generation of chip select signals.

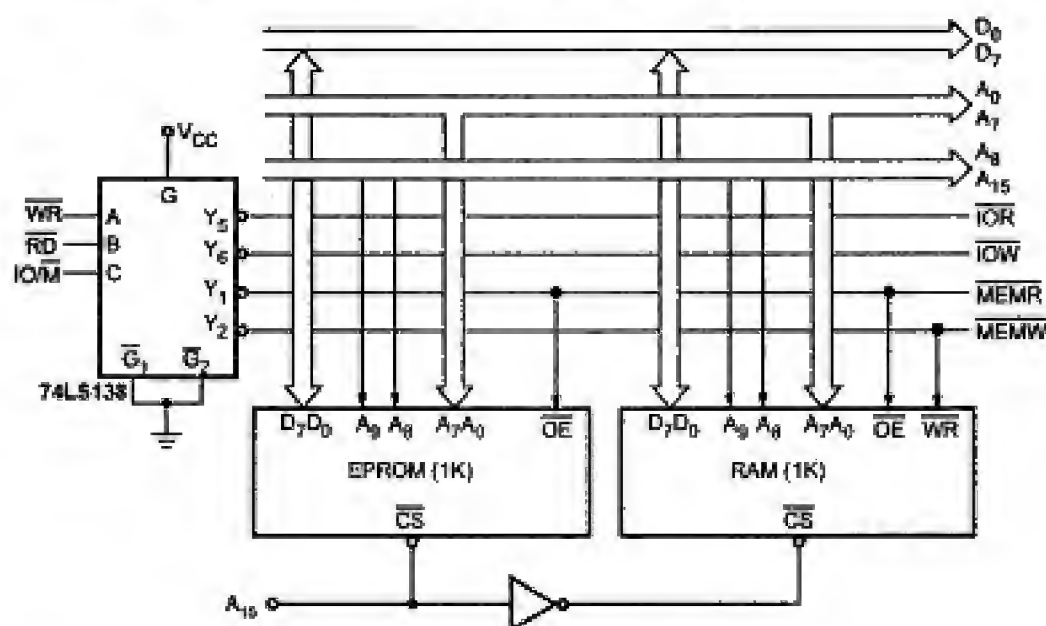


Fig. 6.18 Linear decoding

Memory Map :

| Memory ICs | A ₁₅ | A ₁₄ | A ₁₃ | A ₁₂ | A ₁₁ | A ₁₀ | A ₉ | A ₈ | A ₇ | A ₆ | A ₅ | A ₄ | A ₃ | A ₂ | A ₁ | A ₀ | Address |
|---------------------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|---------|
| Starting address of EPROM | 0 | X | X | X | X | X | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0000H |
| End address of EPROM | 0 | X | X | X | X | X | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 03FFH |
| Starting address of RAM | 1 | X | X | X | X | X | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 8000H |
| End address of RAM | 1 | X | X | X | X | X | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 83FFH |

Table 6.4

| Sr. No. | Full Address Decoding | Partial Address Decoding |
|---------|--|--|
| 1. | All higher address lines are decoded to select the memory or I/O device. | Few higher address lines are decoded to select the memory or I/O device. |
| 2. | More hardware is required to design decoding logic. | Hardware required to design decoding logic is less and sometimes it can be eliminated. |
| 3. | Higher cost for decoding circuit. | Less cost for decoding circuit. |
| 4. | No multiple addresses. | It has a disadvantage of multiple addresses (shadow addresses). |
| 5. | Used in large systems. | Used in small systems. |

Table 6.5

6.4.2 Interfacing Examples

➡ **Example 6.3 :** Design memory system for the 8085 microprocessor such that it should contain 8 Kbyte of EPROM (Erasable Programmable Read Only Memory) and 8 Kbyte of RAM (Read/Write Memory).

Solution : Fig. 6.19 shows the desired memory system using IC 2764 (8K) EPROM and 6264 (8K) RAM. Memory requires 13 address lines (A_0-A_{12}) since $2^{13} = 8K$. The remaining address lines ($A_{13}-A_{15}$) are decoded to generate chip select (\overline{CS}) signals. IC 74LS138 is used as decoder. When ($A_{15}-A_{13}$) address lines are zero, the Y_0 output of decoder goes low and selects the EPROM. This means that $A_{15}-A_{13}$ address lines must be zero to read data from EPROM. The address lines $A_{12}-A_0$ select the particular memory location in the EPROM when $A_{15}-A_{13}$ lines are zero. Similarly, when address lines $A_{15}-A_{13}$ are 001, the Y_1 output of decoder goes low and selects the RAM. The Table 6.6 shows the memory map for the designed circuit.

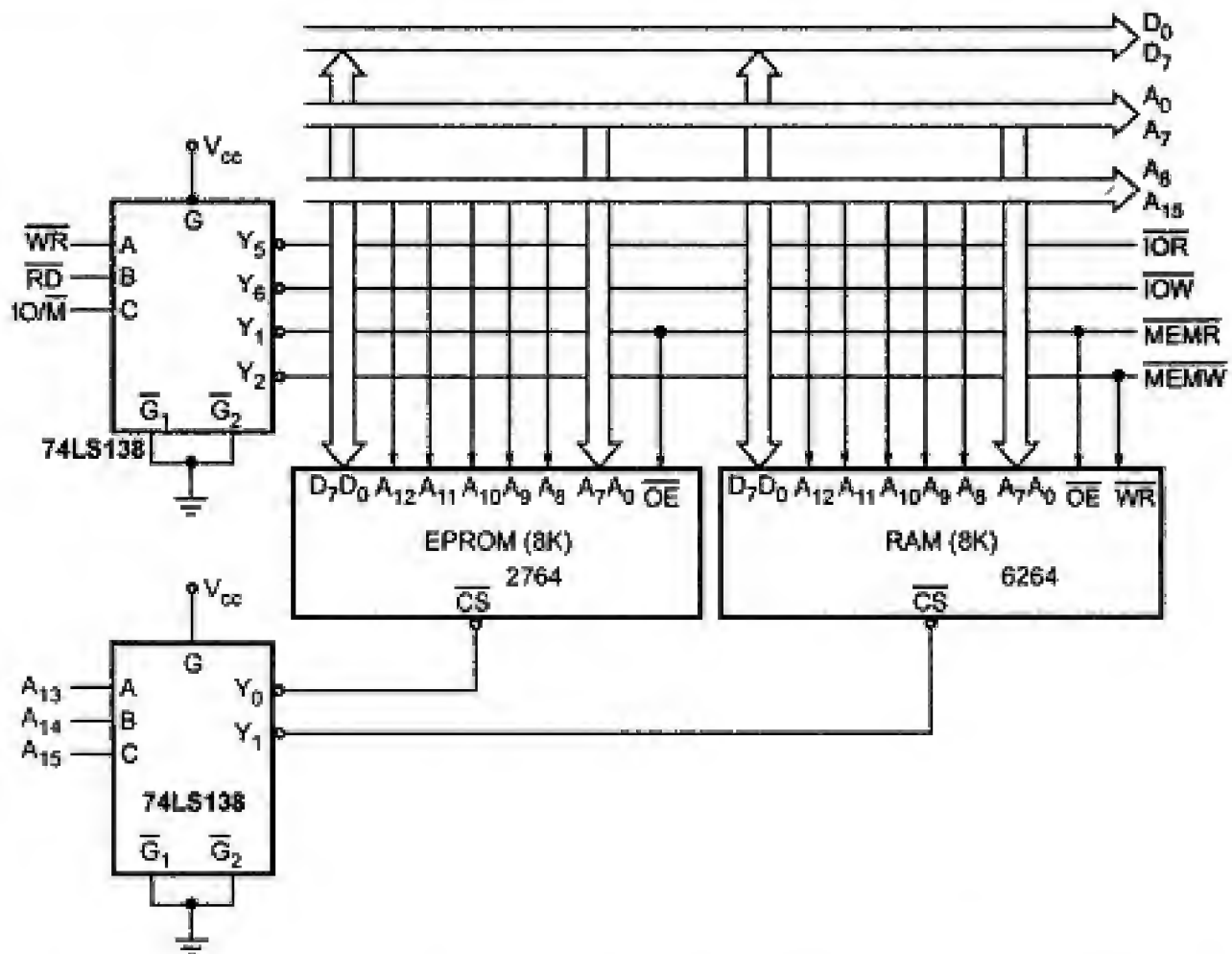


Fig. 6.19 Memory system using IC 2764 (8K) EPROM and 6264 (8K) RAM

Memory Map :

| Memory ICs | A ₁₅ | A ₁₄ | A ₁₃ | A ₁₂ | A ₁₁ | A ₁₀ | A ₉ | A ₈ | A ₇ | A ₆ | A ₅ | A ₄ | A ₃ | A ₂ | A ₁ | A ₀ | Address |
|---------------------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|---------|
| Starting address of EPROM | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0000H |
| End address of EPROM | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1FFFH |
| Starting address of RAM | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2000H |
| End address of RAM | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 3FFFH |

Table 6.6 Memory map

➡ **Example 6.4 :** Design a microprocessor system for the 8085 microprocessor such that it should contain 16 Kbyte of EPROM and 4 Kbyte of RAM using two 8 Kbyte EPROMs (2764) and two 2 Kbyte RAMs (6116).

Solution : Fig. 6.20 shows the desired memory system using two (8K × 8) EPROM and two (2K × 8) RAMs. EPROM memory is 8K, so it requires 13 address lines (A₁₂ - A₀) whereas RAM memory is 2K, so it requires 11 address lines (A₁₀ - A₀). The remaining higher address lines (A₁₅ - A₁₃) are used to generate chip-select (\overline{CS}) signals. Table 6.7 shows the memory map for the designed circuit. (See Fig. 6.20 on next page)

Memory Map :

| Memory ICs | A ₁₅ | A ₁₄ | A ₁₃ | A ₁₂ | A ₁₁ | A ₁₀ | A ₉ | A ₈ | A ₇ | A ₆ | A ₅ | A ₄ | A ₃ | A ₂ | A ₁ | A ₀ | Address |
|-----------------------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|---------|
| Starting Address of EPROM 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0000H |
| End Address of EPROM 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1FFFH |
| Starting Address of EPROM 2 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2000H |
| End Address of EPROM 2 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 3FFFH |
| Starting Address of RAM 1 | 0 | 1 | 0 | X | X | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4000H |
| End Address of RAM 1 | 0 | 1 | 0 | X | X | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 47FFH |
| Starting Address of RAM 2 | 0 | 1 | 1 | X | X | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 6000H |
| End Address of RAM 2 | 0 | 1 | 1 | X | X | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 67FFH |

Table 6.7 Memory map

➡ **Example 6.5 :** Interface 2 Kbyte RAM to 8085 using 2114 (1K × 4) chips, 74LS138 decoder and full address decoding and give the address map (memory map).

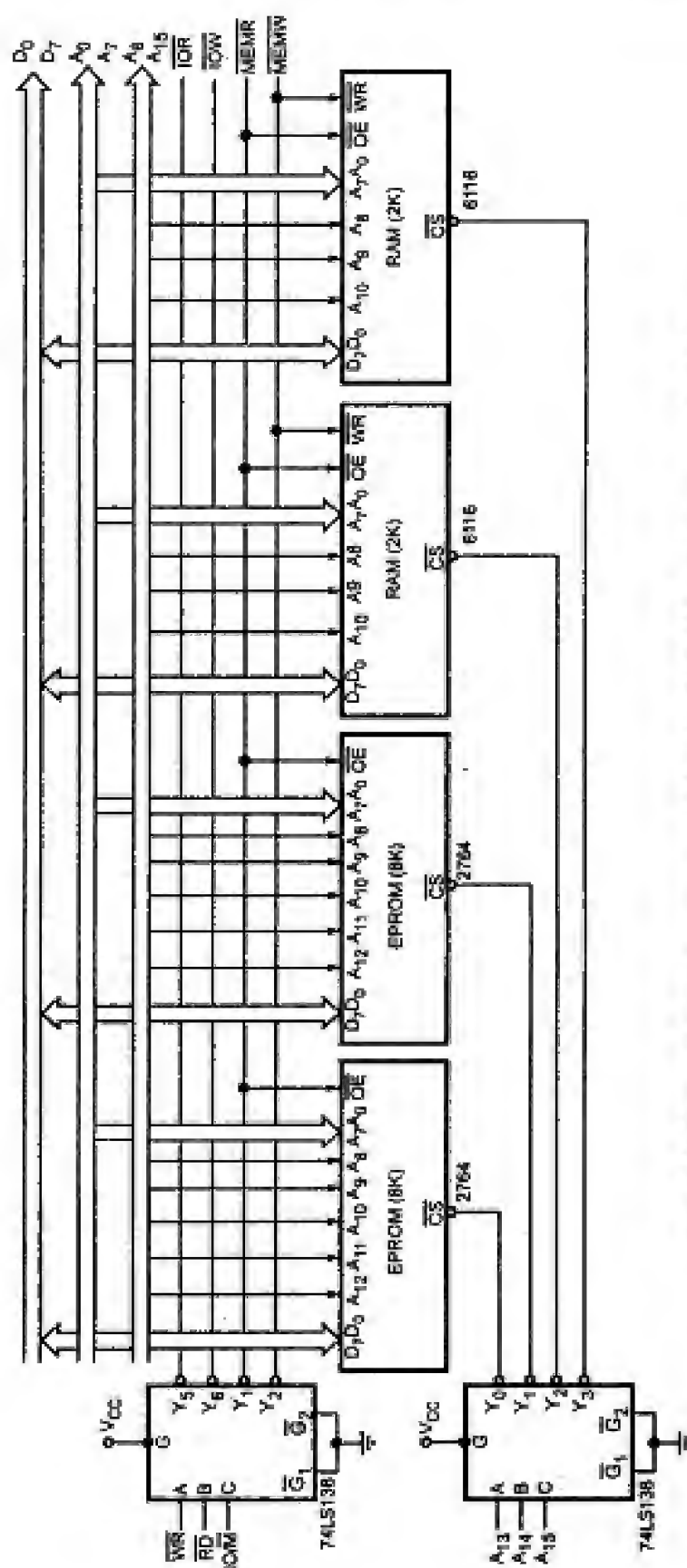


Fig. 6.20 Memory system using two (8K x 8) EPROMs and two (2K x 8) RAMs

Solution : 2114 RAM is $1K \times 4$ i.e. it has 1K (1024) memory locations, each of which is of 4 bits. 8085 is an 8 bit processor. To interface byte RAM, we require two nibble wide RAMs, connected together to form byte wide RAM. To form $2K \times 8$ RAM we require two sets of $1K \times 4 + 1K \times 4$ RAM chips. So in all we require four $1K \times 4$ RAM chips. Fig. 6.21 (See on next page) shows the interface. The IC 74LS138 is used to generate chip select (\overline{CS}) signals. Table 6.8 shows the memory map.

Memory Map :

| Memory ICs | A ₁₅ | A ₁₄ | A ₁₃ | A ₁₂ | A ₁₁ | A ₁₀ | A ₉ | A ₈ | A ₇ | A ₆ | A ₅ | A ₄ | A ₃ | A ₂ | A ₁ | A ₀ | Address |
|---------------------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|---------|
| Starting Address of RAM 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0000H |
| End Address of RAM 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 03FFH |
| Starting Address of RAM 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0000H |
| End Address of RAM 2 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 03FFH |
| Starting Address of RAM 3 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2000H |
| End Address of RAM 3 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 23FFH |
| Starting Address of RAM 4 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2000H |
| End Address of RAM 4 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 23FFH |

Table 6.8 Memory map

➡ **Example 6.6 :** Design a microprocessor system for the 8085 microprocessor such that it should contain 2 Kbyte of EPROM and 2 Kbyte of RAM with starting addresses 0000H and 6000H respectively.

Solution : Fig. 6.22 (See on page no.6-33) shows the desired memory system using 2 Kbyte EPROM and 2 Kbyte RAM. Both EPROM and RAM are 2K, so they require 11 address lines ($A_{10} - A_0$). The remaining higher address lines ($A_{15} - A_{11}$) are used to generate chip select (\overline{CS}) signals.

The chip selection logic is designed to have starting address of EPROM, 0000H and starting address of RAM, 6000H. This is implemented by selecting EPROM only when higher address lines ($A_{15} - A_{11}$) are all zero, and selecting RAM only when higher address lines ($A_{15} - A_{11}$) are 01100 (Binary). The Table 6.9 shows the memory map for the designed circuit.

Memory Map :

| Memory ICs | A ₁₅ | A ₁₄ | A ₁₃ | A ₁₂ | A ₁₁ | A ₁₀ | A ₉ | A ₈ | A ₇ | A ₆ | A ₅ | A ₄ | A ₃ | A ₂ | A ₁ | A ₀ | Address |
|---------------------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|---------|
| Starting address of EPROM | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0000H |
| End address of EPROM | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 07FFH |
| Starting address of RAM | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 6000H |
| End address of RAM | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 67FFH |

Table 6.9 Memory map

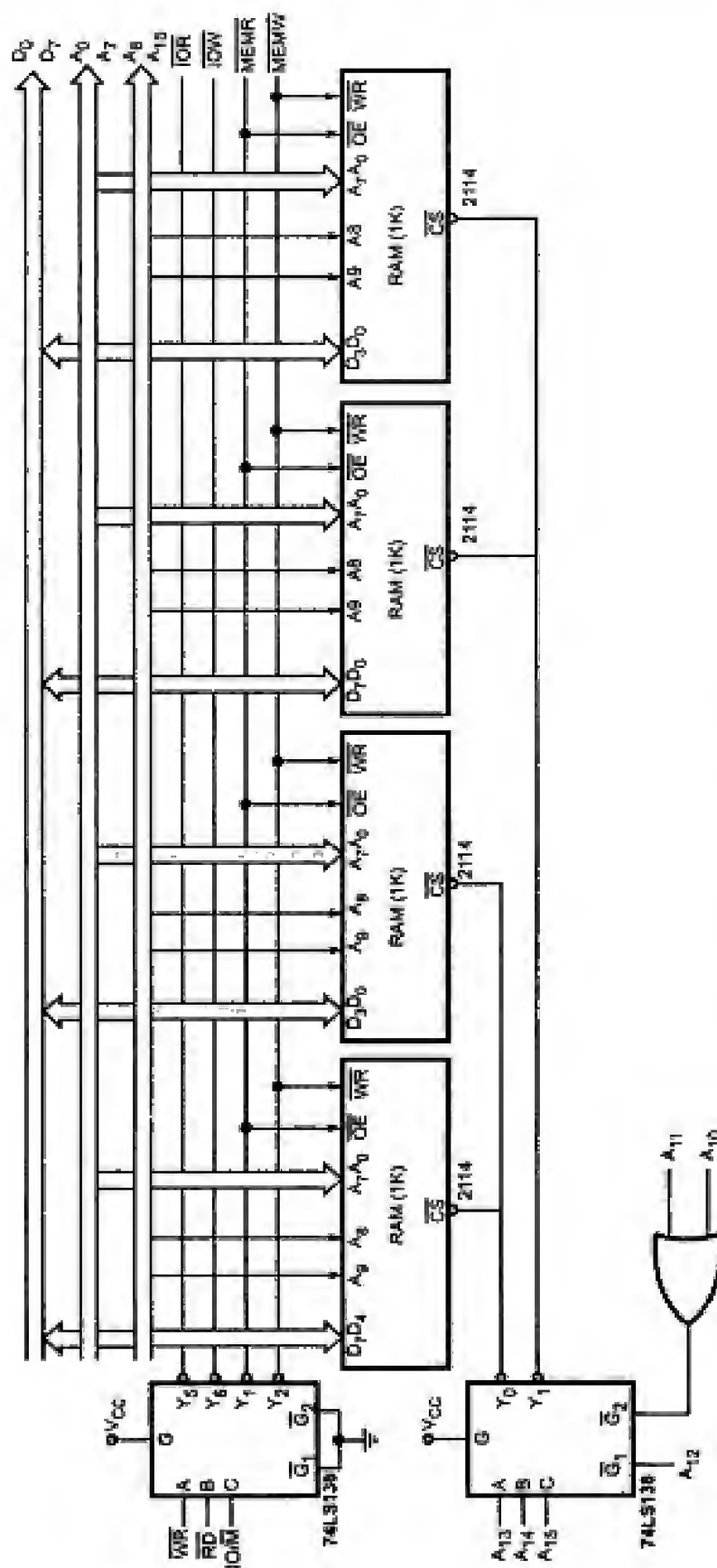


Fig. 6.21 Memory system using 2114 (1K × 4) chips

➡ **Example 6.7 :** Design a microprocessor system for the 8085 microprocessor such that it should contain 4 Kbyte of EPROM and 2 Kbyte of RAM using two 2 Kbytes of EPROMs and two 1 Kbytes RAM. Draw the complete interfacing diagram with buffers, latches, and chip select logic used.

Solution : Fig. 6.23 shows the desired memory system using two 2 Kbyte EPROMs and two 1 Kbyte RAMs.

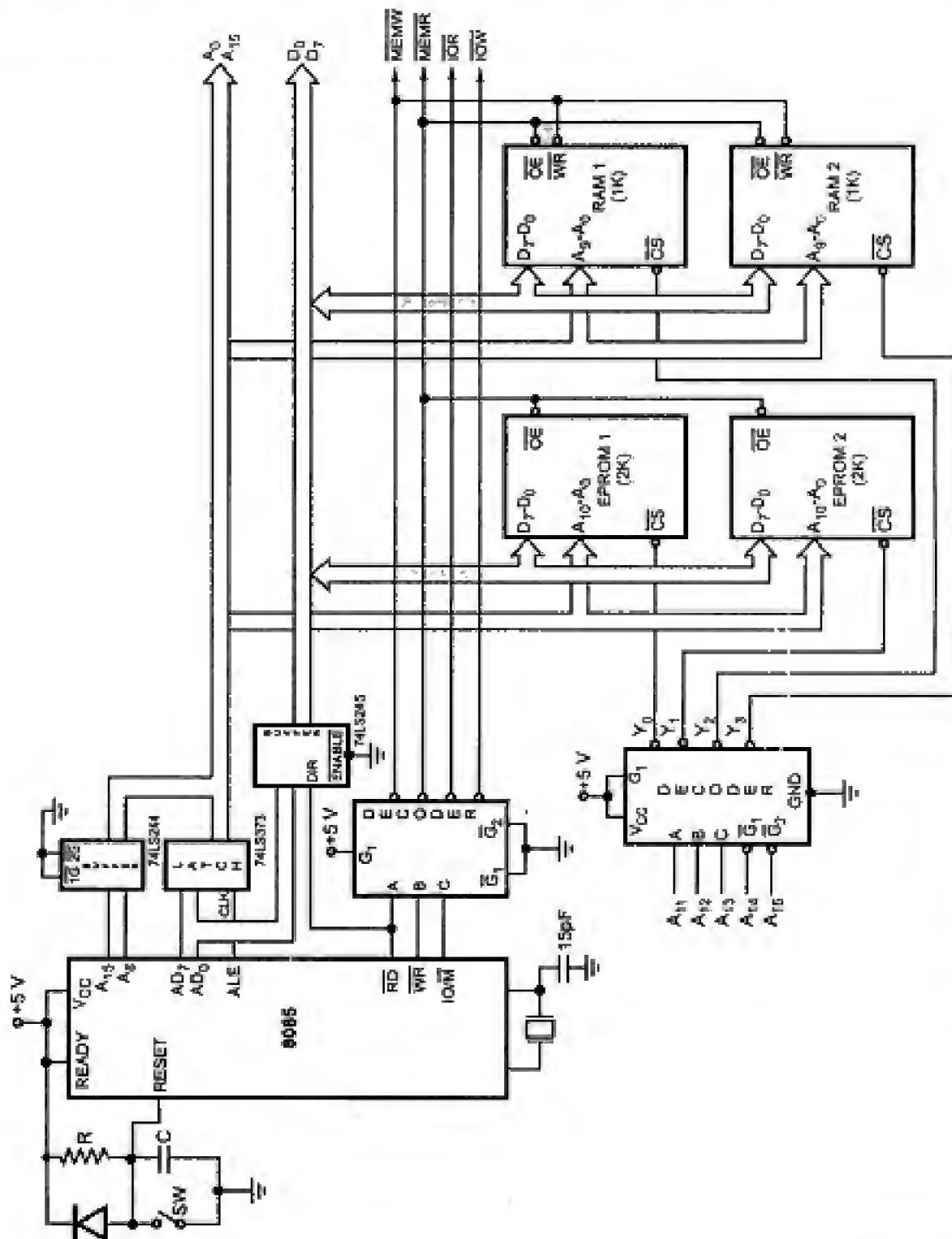


Fig. 6.23 Memory system using 2 Kbyte EPROMs and two 1 Kbyte RAMs

The uni-directional buffer is used to drive the high order address bus and bi-directional buffer is used to drive the data bus. The direction pin of the bi-directional buffer is controlled by the \overline{RD} signal from the microprocessor. EPROM memory is 2 Kbyte, so it requires 11 address lines (A_{10} - A_0) whereas RAM memory is 1K, so it requires 10 address lines (A_9 - A_0). The remaining higher address lines (A_{15} - A_{11}) are used to generate chip-select (\overline{CS}) signals. Table 6.10 shows the memory map for the designed circuit.

Memory Map :

| Memory ICs | A_{15} | A_{14} | A_{13} | A_{12} | A_{11} | A_{10} | A_9 | A_8 | A_7 | A_6 | A_5 | A_4 | A_3 | A_2 | A_1 | A_0 | Address |
|-----------------------------|----------|----------|----------|----------|----------|----------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|---------|
| Starting Address of EPROM 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0000H |
| End Address of EPROM 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 07FFH |
| Starting Address of EPROM 2 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0800H |
| End Address of EPROM 2 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0FFFH |
| Starting Address of RAM 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1000H |
| End Address of RAM 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 13FFH |
| Starting Address of RAM 2 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1800H |
| End Address of RAM 2 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1BFFH |

Table 6.10 Memory map

➡ **Example 6.8 :** Design a microprocessor system for the 8085 microprocessor such that it should contain 8 Kbyte of EPROM and 8 Kbyte of RAM. Use linear addressing technique and give the detailed address map.

Solution : Fig. 6.24 shows the desired memory system using 8 Kbyte EPROM and 8 Kbyte RAM. Both EPROM and RAM are 8K, so they require 13 address lines (A_{12} - A_0). As problem says to design microprocessor system using linear addressing technique, A_{15} address line is used to generate chip select (\overline{CS}) signals. When A_{15} address line is low, it selects EPROM and when it is high, it selects RAM. The Table 6.11 shows the memory map for the designed circuit.

Memory Map :

| Memory ICs | A_{15} | A_{14} | A_{13} | A_{12} | A_{11} | A_{10} | A_9 | A_8 | A_7 | A_6 | A_5 | A_4 | A_3 | A_2 | A_1 | A_0 | Address |
|---------------------------|----------|----------|----------|----------|----------|----------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|---------|
| Starting address of EPROM | 0 | X | X | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0000H |
| End address of EPROM | 0 | X | X | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1FFFH |
| Starting address of RAM | 1 | X | X | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 8000H |
| End address of RAM | 1 | X | X | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 9FFFH |

Table 6.11 Memory map

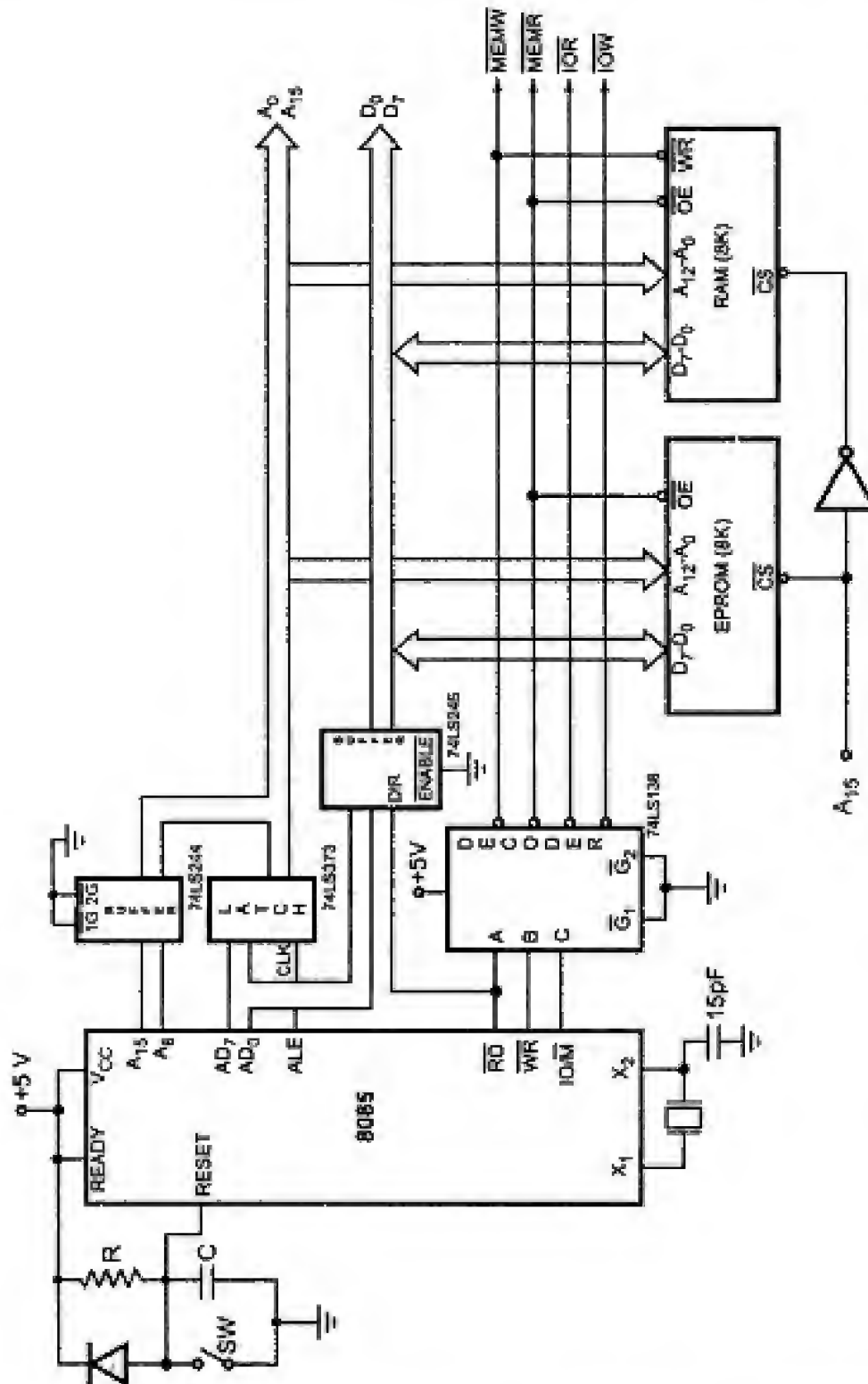


Fig. 6.24 Memory system using 8 Kbyte EPROM and 8 Kbyte RAM with linear address decoding

Note : X represents the don't care condition and addresses are written taking don't cares as zeroes.

Review Questions

1. What are the important functions of I/O system ?
2. Draw and explain generic model of computer showing I/O system.
3. Give the major requirements for an I/O module.
4. Draw and explain the block diagram of I/O system.
5. What do you mean by input port and output port ?
6. Write a short note on
 - a) Programmed I/O
 - b) Interrupt driven I/O
 - c) DMA controlled I/O
7. Explain I/O mapped I/O interfacing technique.
8. Explain memory mapped I/O interfacing technique.
9. Compare I/O mapped and memory mapped I/O techniques.
10. Explain with the help of neat diagram interfacing of input device.
11. Explain with the help of neat diagram interfacing of output device.
12. List the important points which must be considered while interfacing memory devices to 8085.
13. Give the comparison between full and partial decoding techniques.
14. Design an 8085 processor based system using 4 Kbytes of EPROM and 4 Kbytes of RAM. EPROM address starts at 0000H and RAM address at 4000H. Draw the organisation of the memory section which is to be interfaced to the 8085 microprocessor. Indicate clearly the address range for each chip. Use standard address decoders. Explain the various schemes of data transfer in microprocessor systems.
15. An 8 KB ROM having a word length of 8 bits is to occupy the first 8 KB of the address space of an 8085A processor. Two 2 KB RAMs having word lengths of 4 bits each are to occupy 2 KB starting from address 4000H. Design address decoding logic.
16. Interface 8 Kbytes of EPROM and 8 Kbytes of RAM to 8085.
17. Design a memory address decoder to interface a 4 KB PROM starting from location 4000H, 16KB EPROM starting from location 0000H and 32 KB RAM starting from location 8000H. One or more of the following available IC's can be used.
PROM IC : 4K \times 8 EPROM IC : 8K \times 8 RAM IC : 16K \times 8



Microcontroller 8051

7.1 Introduction

To make a complete microcomputer system, only microprocessor is not sufficient. It is necessary to add other peripherals such as read only memory (ROM), read/write memory (RAM), decoders, drivers, number of input/output devices to make a complete microcomputer system. In addition, special purpose devices, such as interrupt controller, programmable timers, programmable I/O devices, DMA controllers may be added to improve the capability and performance and flexibility of a microcomputer system.

The key feature of microprocessor based computer system is that it is possible to design a system with a great flexibility. It is possible to configure a system as large system or small system by adding suitable peripherals.

On the other hand, the microcontroller incorporates all the features that are found in microprocessor. However, it has also added features to make a complete microcomputer system on its own. The microcontroller has built-in ROM, RAM, parallel I/O, serial I/O, counters and a clock circuit.

The microcontroller has on-chip (built-in) peripheral devices. These on-chip peripherals make it possible to have single chip microcomputer system. There are few more advantages of built-in peripherals :

- Built-in peripherals have smaller access times hence speed is more.
- Hardware reduces due to single chip microcomputer system.
- Less hardware, reduces PCB size and increases reliability of the system.

Comparison between Microprocessor and Microcontroller

We have discussed what is a microprocessor and a microcontroller. Let us see the points of differences between them.

| No. | Microprocessor | Microcontroller |
|-----|---|---|
| 1. | Microprocessor contains ALU, control unit (clock and timing circuit), different register and interrupt circuit. | Microcontroller contains microprocessor, memory (ROM and RAM), I/O interfacing circuit and peripheral devices such as A/D converter, serial I/O, timer etc. |
| 2. | It has many instructions to move data between memory and CPU. | It has one or two instructions to move data between memory and CPU. |
| 3. | It has one or two bit handling instructions. | It has many bit handling instructions. |
| 4. | Access times for memory and I/O devices are more. | Less access times for built-in memory and I/O devices. |
| 5. | Microprocessor based system requires more hardware. | Microcontroller based system requires less hardware reducing PCB size and increasing the reliability. |
| 6. | Microprocessor based system is more flexible in design point of view. | Less flexible in design point of view. |
| 7. | It has single memory map for data and code. | It has separate memory map for data and code. |
| 8. | Less number of pins are multifunctioned. | More number pins are multifunctioned. |

The 8051 is an 8-bit microcontroller designed by Intel. It was optimized for 8-bit mathematical and single bit Boolean operations. Its family-MCS-51 includes 8031, 8051 and 8751 microcontrollers. The Table 7.1 gives the summary of MCS-51 microcontrollers.

| Device | Internal memory | | Timer / | Interrupts |
|----------|-----------------|-------------|----------------|------------|
| | Program | Data | Event counters | |
| 8052AH | 8 K × 8 ROM | 256 × 8 RAM | 3 × 16-Bit | 6 |
| 8051AH | 4 K × 8 ROM | 128 × 8 RAM | 2 × 16-Bit | 5 |
| 8051 | 4 K × 8 ROM | 128 × 8 RAM | 2 × 16-Bit | 5 |
| 8032AH | none | 256 × 8 RAM | 2 × 16-Bit | 6 |
| 8031AH | none | 128 × 8 RAM | 2 × 16-Bit | 5 |
| 8031 | none | 128 × 8 RAM | 2 × 16-Bit | 5 |
| 8751H | 4 K × 8 EPROM | 128 × 8 RAM | 2 × 16-Bit | 5 |
| 8751H-12 | 4 K × 8 EPROM | 128 × 8 RAM | 2 × 16-Bit | 5 |

Table 7.1 MCS-51 family

In this chapter we are going to see features and the internal hardware details (architecture) of 8051 microcontroller.

7.2 Features of 8051

The features of the 8051 family are as follows :

1. 4096 bytes on-chip program memory.
2. 128 bytes on-chip data memory.
3. Four register banks.
4. 128 user-defined software flags.
5. 64 kilobytes each program and external RAM addressability.
6. One microsecond instruction cycle with 12 MHz crystal.
7. 32 bidirectional I/O lines organized as four 8-bit ports (16 lines on 8031).
8. Multiple mode, high-speed programmable serial port.
9. Two multiple mode, 16-bit Timers/Counters.
10. Two-level prioritized interrupt structure.
11. Full depth stack for subroutine return linkage and data storage.
12. Direct byte and bit addressability.
13. Binary or decimal arithmetic.
14. Signed-overflow detection and parity computation.
15. Hardware multiple and divide in 4 μ sec.
16. Integrated Boolean Processor for control applications.
17. Upwardly compatible with existing 8084 software.

7.3 8051 Microcontroller Hardware

The Fig. 7.1 shows the internal block diagram of 8051. It consists of a CPU, two kinds of memory sections (data memory - RAM and program memory - EPROM/ROM), input/output ports, special function registers and control logic needed for a variety of peripheral functions. These elements communicate through an eight bit data bus which runs throughout the chip referred as internal data bus. This bus is buffered to the outside world through an I/O port when memory or I/O expansion is desired.

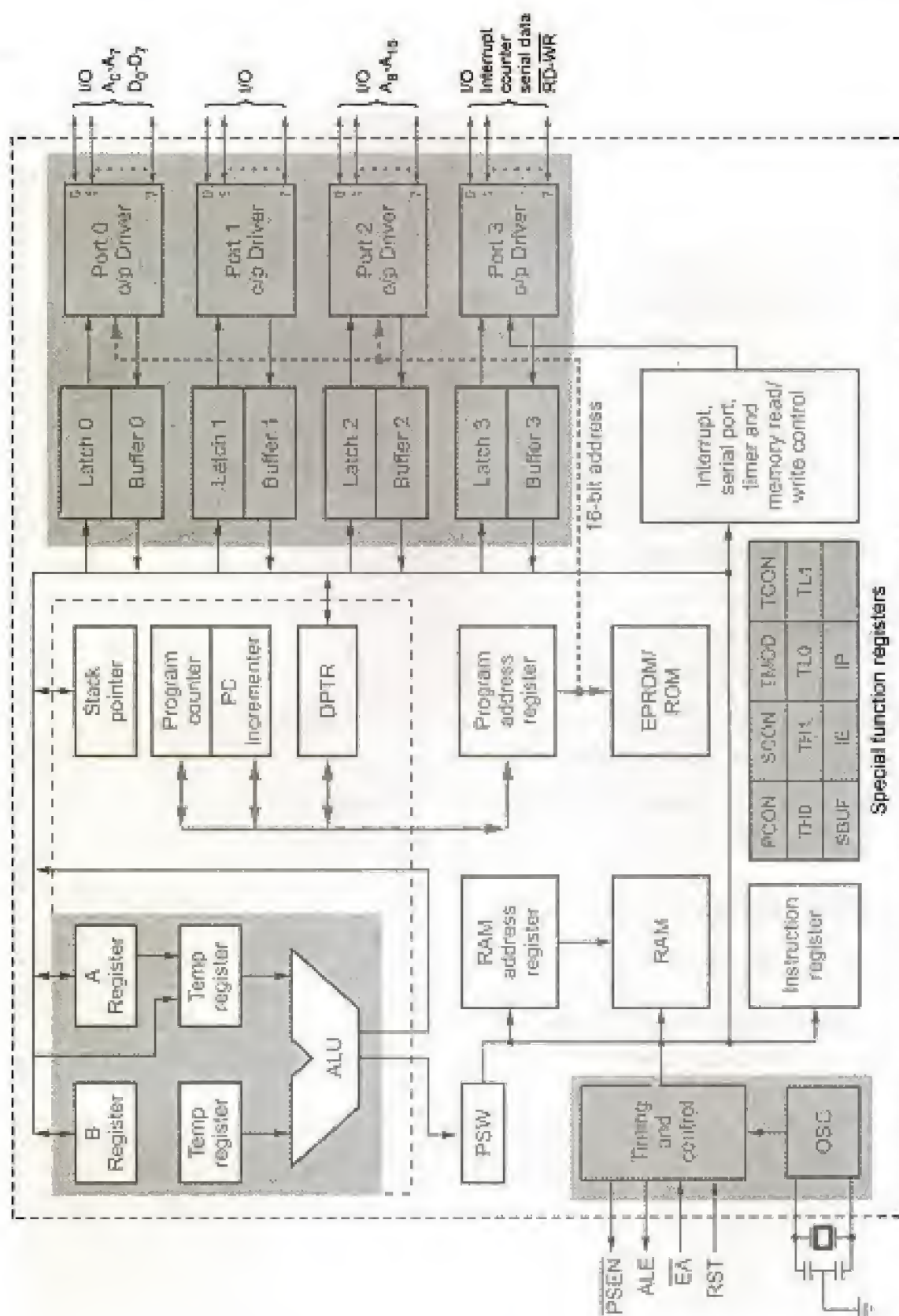


Fig. 7.1 Block diagram of 8051

7.3.1 Pin-out of 8051

The 8051 is packaged in a 40-pin DIP. The Fig. 7.2 shows the pin diagram of 8051. It is important to note that many pins of 8051 are used for more than one function. The alternative functions of pins are shown in bold letters.

Fig. 7.2 shows the pin diagram of 8051.

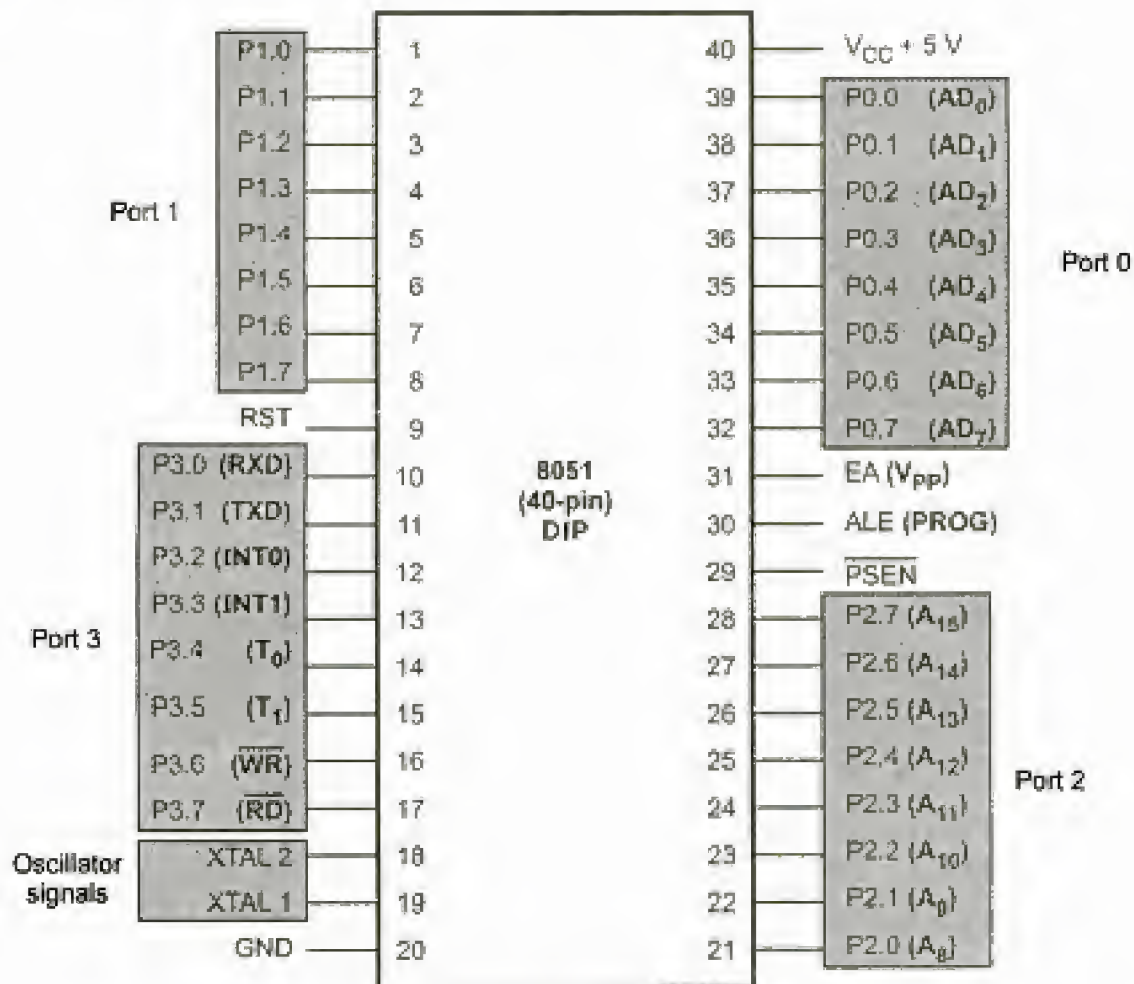


Fig. 7.2 Pin-out of 8051

Table 7.2 gives the pin-description. The function of all pins of 8051 are explained below.

The 8051 has 32 I/O pins configured as four eight-bit parallel ports (P0, P1, P2 and P3). All four ports are bidirectional i.e. each pin will be configured as input or output (or both). All port-pins are multiplexed except the pins of port 1. Each port consists of a latch, an output driver and an input buffer.

Port 0 (Pins 32 - 39)

Port 0 pins can be used as I/O pins. The output drives and input buffers of port 0 are used to access external memory. Port 0 outputs the low order byte of the external memory address, time multiplexed with the data being written or read. Thus, port 0 can be used as a multiplexed address/data bus.

Port 1 (Pins 1 - 8)

Port 1 pins can be used only as I/O pins.

Port 2 (Pins 21 - 28)

The output drives of port 2 are used to access external memory. Port 2 outputs the high order byte of the external memory address when the address is 16-bits wide. Otherwise, port 2 is used as an I/O port.

Port 3 (Pins 10 - 17)

All port pins of port 3 are multifunctional. They have special functions as shown below including two external interrupts, two counter inputs, two special data lines and two timing control strobes.

| Symbol | Position | Name and significance |
|-------------------|----------|---|
| \overline{RD} | P3.7 | Read data control output. Active low pulse generated by hardware when external data memory is read. |
| \overline{WR} | P3.6 | Write data control output. Active low pulse generated by hardware when external data memory is written. |
| T1 | P3.5 | Timer/counter 1 external input or test pin. |
| T0 | P3.4 | Timer/counter 0 external input or test pin. |
| $\overline{INT1}$ | P3.3 | Interrupt 1 input pin. Low-level or falling-edge triggered. |
| $\overline{INT0}$ | P3.2 | Interrupt 0 input pin. Low-level or falling-edge triggered. |
| TXD | P3.1 | Transmit Data pin for serial port in UART mode. Clock output in shift register mode. |
| RXD | P3.0 | Receive Data pin for serial port in UART mode. Data I/O pin in shift register mode. |

Table 7.2

Power-supply pins V_{CC} (Pin 40) and V_{SS} (Pin 20)

8051 operates on d.c. power supply of +5 V with respect to ground. The +5 V is to be connected to pins V_{CC} and ground to pin V_{SS} with rated power supply current of 125 mA.

Oscillator pins XTAL2 (Pin 18) and XTAL 1 (Pin 19)

For generating an internal clock signal, the external oscillator is connected at these two pins.

ALE (Address Latch Enable, Pin 30)

AD_0 to AD_7 lines are multiplexed. To demultiplex these lines and for obtaining lower half of an address, an external latch and ALE signal of 8051 is used.

RST (Reset, Pin 9)

This pin is used to reset 8051. For proper reset operation, reset signal must be held high atleast for two machine cycles, while oscillator is running.

 \overline{PSEN} (Program Store Enable, Pin 29)

It is the active low output control signal used to activate the enable signal of the external ROM/EPROM. It is activated every six oscillator periods while reading the external memory. Thus, this signal acts as the read strobe to external program memory.

 \overline{EA} (External Access, Pin 31)

When the \overline{EA} pin is high (connected to V_{CC}), program fetches to addresses 0000H through 0FFFH are directed to the internal ROM and program fetches to addresses 1000H through FFFFH are directed to external ROM/EPROM. When \overline{EA} is low (grounded), all addresses (0000H to FFFFH) fetched by program are directed to the external ROM/EPROM.

7.3.2 Central Processing Unit (CPU)

The CPU of 8051 consists of eight-bit arithmetic and logic unit with associated registers like A, B, PSW, SP, the sixteen bit program counter and "Data Pointer" (DPTR) registers. Alongwith these registers it has a set of special function registers. Along with these registers it has a set of special function registers.

The 8051's ALU can perform arithmetic and logic functions on eight bit variables. The arithmetic unit can perform addition, subtraction, multiplication and division. The logic unit can perform logical operations such as AND, OR and Exclusive-OR, as well as rotate, clear, and complement. The ALU also looks after the branching decisions. An important and unique feature of the 8051 architecture is that the ALU can also manipulate one bit as well as eight-bit data types. Individual bits may be set, cleared, complemented, moved, tested and used in logic computation.

7.3.3 Internal RAM

The 8051 has 128-byte internal RAM. It is accessed using RAM address register. The Fig. 7.3 shows the organisation of internal RAM. As shown in the Fig. 7.3, internal RAM of 8051 is organised into three distinct areas :

- Working registers
- Bit addressable
- General purpose

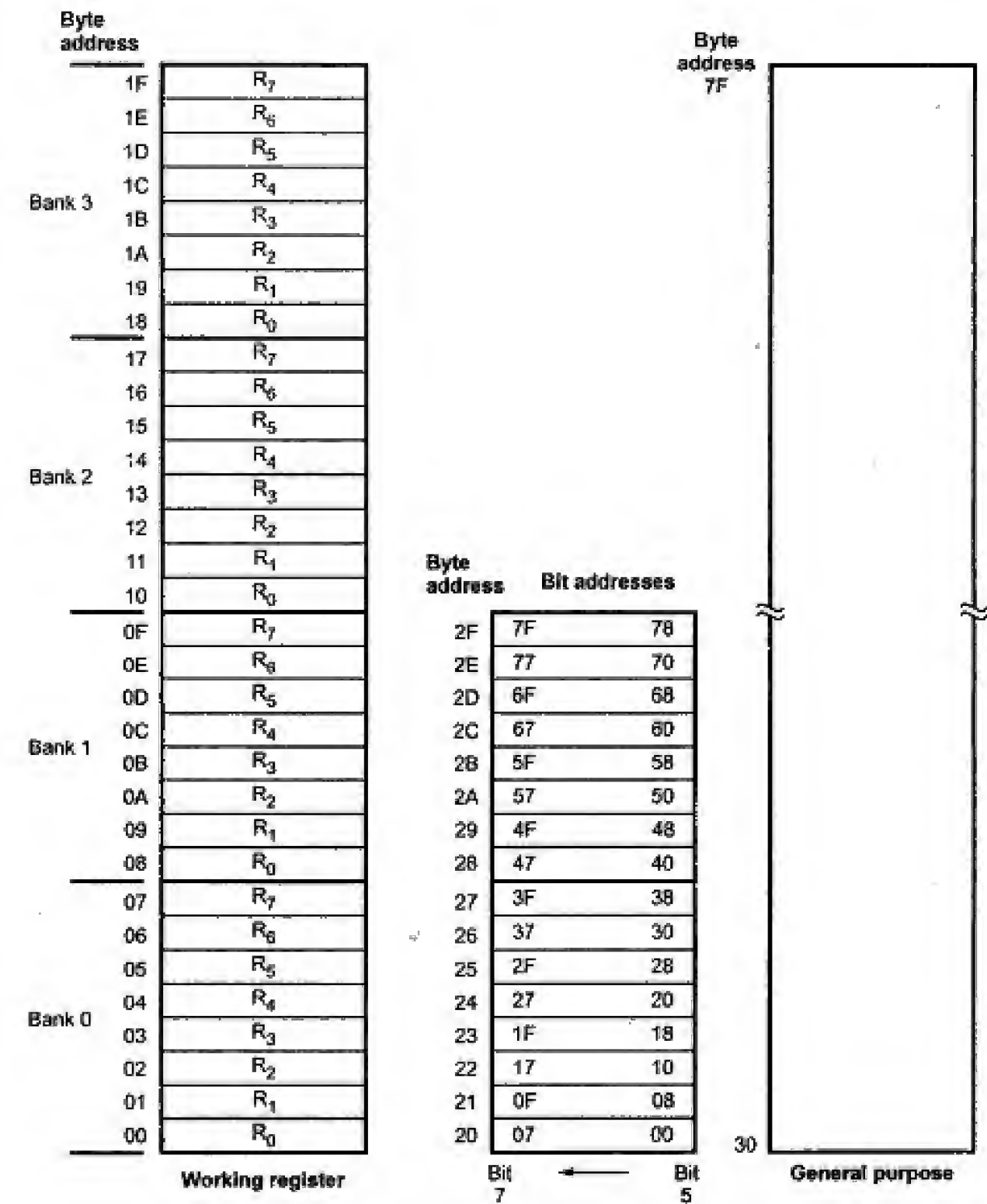
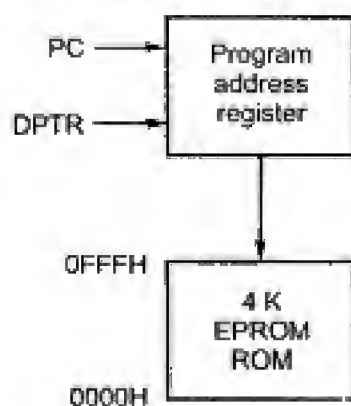


Fig. 7.3 Organisation of internal RAM of 8051

1. First thirtytwo bytes from address 00H to 1FH of internal RAM constitute 32 working registers. They are organised into four banks of eight registers each. The four register banks are numbered 0 to 3 and are consists of eight registers named R_0 to R_7 . Each register can be addressed by name or by its RAM address. Only one register bank is in use at a time. Bits RS_0 and RS_1 in the PSW determine which bank of registers is currently in use. Register banks when not selected can be used as general purpose RAM. On reset, the bank 0 is selected.
2. The 8051 provides 16 bytes of a bit-addressable area. It occupies RAM byte addresses from 20H to 2FH, forming a total of 128 (16×8) addressable bits. An addressable bit may be specified by its bit address of 00H to 7FH, or 8 bits may form any byte address from 20H to 2FH. For example, bit address 4EH refers bit 6 of the byte address 29H.
3. The RAM area above bit addressable area from 30H to 7FH is called general purpose RAM. It is addressable as byte.

7.3.4 Internal ROM



The 8051 has 4 K byte of internal ROM with address space from 0000H to 0FFFH. It is programmed by manufacturer when the chip is built. This part cannot be erased or altered after fabrication. This is used to store final version of the program.

It is accessed using program address register. The program addresses higher than 0FFFH, which exceed the internal ROM capacity will cause the 8051 to automatically fetch code bytes from external program memory. However, code bytes can also be fetched exclusively from an external memory addresses 0000H to FFFFH, by connecting the

external access pin (EA) to ground.

7.3.5 Input/Output Ports

The 8051 has 32 I/O pins configured as four eight-bit parallel ports (P0, P1, P2, and P3). All four ports are bidirectional, i.e. each pin will be configured as input or output (or both) under software control. Each port consists of a latch, an output driver, and an input buffer.

The output drives of Ports 0 and 2 and the input buffers of Port 0, are used to access external memory. As mentioned earlier, Port 0 outputs the low order byte of the external memory address, time multiplexed with the data being written or read, and Port 2 outputs the high order byte of the external memory address when the address is 16-bits wide. Otherwise Port 2 gives the contents of special function register P2.

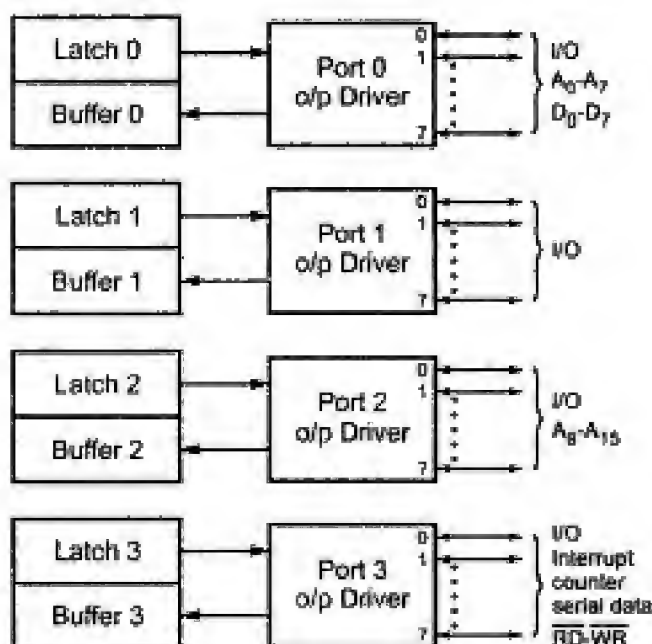


Fig. 7.4 I/O Ports

All port pins of Port 3 are multifunctional. They have special functions as shown below including two external interrupts, two counter inputs, two special data lines and two timing control strobes.

| Symbol | Position | Name and significance |
|-------------------|----------|---|
| \overline{RD} | P3.7 | Read data control output. Active low pulse generated by hardware when external data memory is read. |
| \overline{WR} | P3.6 | Write data control output. Active low pulse generated by hardware when external data memory is written. |
| T1 | P3.5 | Timer/counter 1 external input or test pin. |
| T0 | P3.4 | Timer/counter 0 external input or test pin. |
| $\overline{INT1}$ | P3.3 | Interrupt 1 input pin. Low-level or falling-edge triggered. |
| $\overline{INT0}$ | P3.2 | Interrupt 0 input pin. Low-level or falling-edge triggered. |
| TXD | P3.1 | Transmit Data pin for serial port in UART mode. Clock output in shift register mode. |
| RXD | P3.0 | Receive Data pin for serial port in UART mode. Data I/O pin in shift register mode. |

7.3.6 Register Set of 8051

7.3.6.1 Register A (Accumulator)

It is an 8-bit register. It holds a source operand and receives the result of the arithmetic instructions (addition, subtraction, multiplication, and division). The accumulator can be the source or destination for logical operations and a number of special data movement instructions, including look-up tables and external RAM expansion. Several

functions apply exclusively to the accumulator : rotate, parity computation , testing for zero and so on.

7.3.6.2 Register B

In addition to accumulator, an 8-bit B-register is available as a general purpose register when it is not being used for the hardware multiply/divide operation.

7.3.6.3 Program Status Word (Flag Register)

Many instructions implicitly or explicitly affect (or are affected by) several status flags, which are grouped together to form the Program Status Word. Fig. 7.5 shows the bit pattern of the program status word. It is an 8-bit word, containing the information as follows.

| | | | | | | | | |
|--|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
| | B ₇ | B ₆ | B ₅ | B ₄ | B ₃ | B ₂ | B ₁ | B ₀ |
| | CY | AC | F0 | RS1 | RS0 | OV | - | P |

CY Bit-7 - Carry flag
 AC Bit-6 - Auxiliary carry flag for BCD operations
 F0 Bit-5 - User defined flag (Flag zero)
 RS1, RS0 Bit-4-3 - Select the working register banks as follows :

| RS1 | RS0 | Bank selection | |
|-----|-----|----------------|--------|
| 0 | 0 | 00H - 07H | Bank 0 |
| 0 | 1 | 08H - 0FH | Bank 1 |
| 1 | 0 | 10H - 17H | Bank 2 |
| 1 | 1 | 18H - 1FH | Bank 3 |

OV Bit-2 - Overflow flag
 - Bit-1 - Reserved
 P Bit-0 - Parity flag (1 = Even parity)

Fig. 7.5 Program status word

7.3.6.4 Stack and Stack Pointer

The stack refers to an area of internal RAM that is used in conjunction with certain opcodes data to store and retrieve data quickly. The stack pointer register is used by the 8051 to hold an internal RAM address that is called **top of stack**. The stack pointer register is 8-bit wide. It is increased **before** data is stored during PUSH and CALL instructions and decremented **after** data is restored during POP and RET instructions. This stack array can reside anywhere in on-chip RAM. The stack pointer is initialized to 07H after a reset. This causes the stack to begin at location 08H. The operation of stack and stack pointer is illustrated in Fig. 7.6.

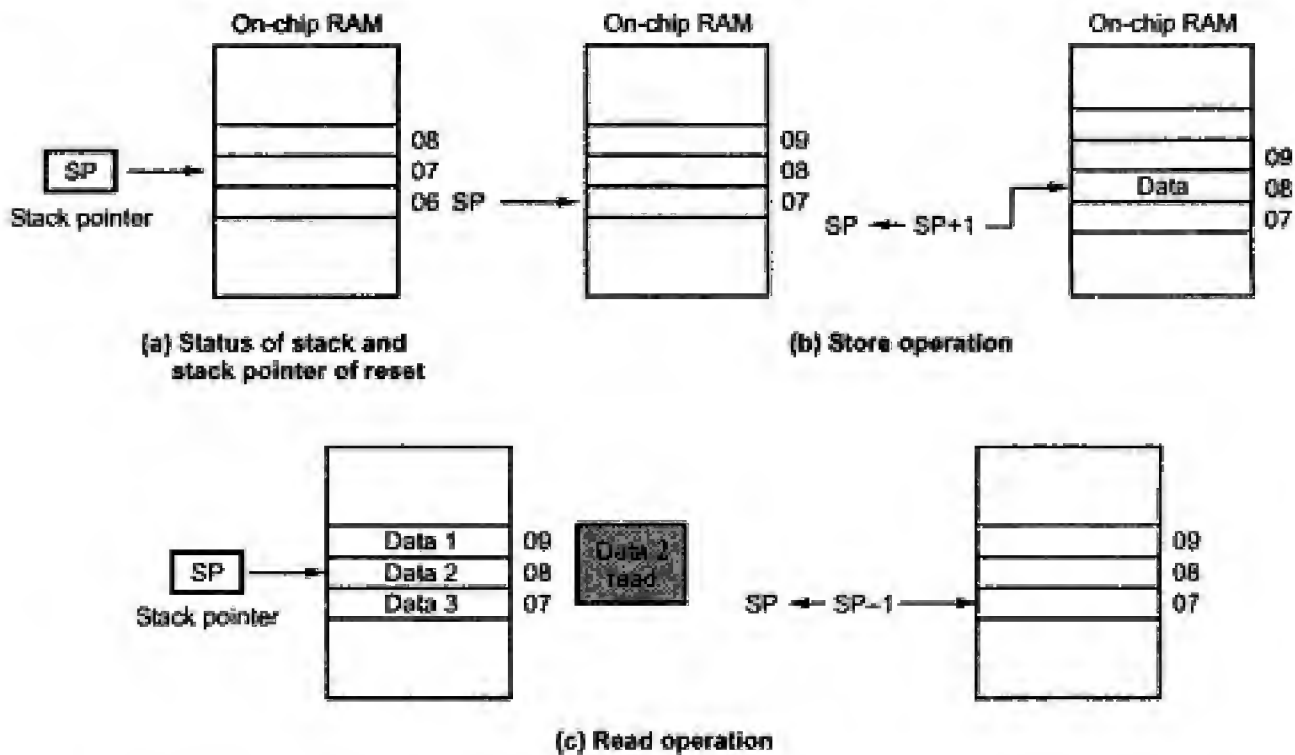
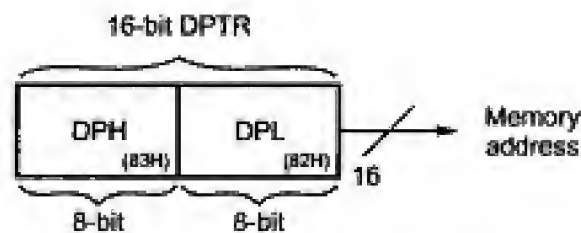


Fig. 7.6

7.3.6.5 Data Pointer (DPTR)

The data pointer (DPTR) consists of a high byte (DPH) and a low byte (DPL). Its function is to hold a 16-bit address. It may be manipulated as a 16-bit data register or as two independent 8-bit registers. It serves as a base register in indirect jumps, lookup table instructions and external data transfer. The DPTR does not have a single internal address; DPH (83H) and DPL (82H) have separate internal addresses.



7.3.6.6 Program Counter

The 8051 has a 16-bit program counter. It is used to hold the address of memory location from which the next instruction is to be fetched. Due to this the width of the program counter decides the maximum program length in bytes. For example, 8051 is 16-bit hence it can address upto 2^{16} bytes (64 K) of memory.

The PC is automatically incremented to point the next instruction in the program sequence after execution of the current instruction. It may also be altered by certain instructions. The PC is the only register that does not have an internal address.

7.3.6.7 Special Function Registers

Unlike other microprocessors in the Intel family, 8051 uses memory mapped I/O through a set of special function registers that are implemented in the address space immediately above the 128 bytes of RAM. Fig. 7.7 shows special function bit addresses. All access to the four I/O ports, the CPU registers, interrupt control registers, the timer/counter, UART and power control are performed through registers between 80H and FFH.

| Direct byte address (MSB) | Bit address (LSB) | | | | | | | | Hardware register symbol |
|---------------------------------|----------------------|----|----|----|----|----|----|----|--------------------------------|
| 0FFH | | | | | | | | | |
| 0F0H | F7 | F6 | F5 | F4 | F3 | F2 | F1 | F0 | B |
| | | | | | | | | | |
| 0E0H | E7 | E6 | E5 | E4 | E3 | E2 | E1 | E0 | ACC |
| | | | | | | | | | |
| 0D0H | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | PSW |
| | | | | | | | | | |
| 0B8H | — | — | — | BC | BB | BA | B9 | B8 | IP |
| | | | | | | | | | |
| 0B0H | B7 | B6 | B5 | B4 | B3 | B2 | B1 | B0 | P3 |
| | | | | | | | | | |
| 0A8H | AF | — | — | AC | AB | AA | A9 | A8 | IE |
| | | | | | | | | | |
| 0A0H | A7 | A6 | A5 | A4 | A3 | A2 | A1 | A0 | P2 |
| | | | | | | | | | |
| 98H | 9F | 9E | 9D | 9C | 9B | 9A | 99 | 98 | SCON |
| | | | | | | | | | |
| 90H | 97 | 96 | 95 | 94 | 93 | 92 | 91 | 90 | P1 |
| | | | | | | | | | |
| 88H | 8F | 8E | 8D | 8C | 8B | 8A | 89 | 88 | TCON |
| | | | | | | | | | |
| 80H | 87 | 86 | 85 | 84 | 83 | 82 | 81 | 80 | P0 |

Fig. 7.7 SFR bit address

Table 7.3 contains a list of all the SFRs and their addresses and their value in binary. Comparing Table 7.3 and Table 7.4 shows that all of the SFRs that are byte and bit addressable are located on the first column of the Table 7.4.

| Symbol | Name | Address | Value in binary |
|-----------|------------------------------|---------|---|
| *ACC | accumulator | 0E0H | 0 0 0 0 0 0 0 0 |
| *B | B Register | 0F0H | 0 0 0 0 0 0 0 0 |
| *PSW | Program Status Word | 0D0H | 0 0 0 0 0 0 0 0 |
| SP | Stack Pointer | 81H | 0 0 0 0 0 1 1 1 |
| DPTR | Data Pointer 2 Bytes | | |
| DPL | Low Byte | 82H | 0 0 0 0 0 0 0 0 |
| DPH | High Byte | 83H | 0 0 0 0 0 0 0 0 |
| *P0 | Port 0 | 80H | 1 1 1 1 1 1 1 1 |
| *P1 | Port 1 | 90H | 1 1 1 1 1 1 1 1 |
| *P2 | Port 2 | 0A0H | 1 1 1 1 1 1 1 1 |
| *P3 | Port 3 | 0B0H | 1 1 1 1 1 1 1 1 |
| *IP | Interrupt Priority Control | 0B8H | 8051 X X X 0 0 0 0 0 8072 X X 0 0 0 0 0 0 |
| *IE | Interrupt Enable Control | 0A8H | 8051 0 X X 0 0 0 0 0 8072 0 X 0 0 0 0 0 0 |
| TMOD | Timer/Counter Mode Control | 89H | 0 0 0 0 0 0 0 0 |
| *TCON | Timer/Counter Control | 88H | 0 0 0 0 0 0 0 0 |
| * + T2CON | Timer/Counter 2 Control | 0C8H | 0 0 0 0 0 0 0 0 |
| TH0 | Timer/Counter 0 High Byte | 8CH | 0 0 0 0 0 0 0 0 |
| TL0 | Timer/Counter 0 Low Byte | 8AH | 0 0 0 0 0 0 0 0 |
| TH1 | Timer/Counter 1 High Byte | 8DH | 0 0 0 0 0 0 0 0 |
| TL1 | Timer/Counter 1 LowByte | 8BH | 0 0 0 0 0 0 0 0 |
| + TH2 | Timer/Counter 2 High Byte | 0CDH | 0 0 0 0 0 0 0 0 |
| + TL2 | Timer/Counter 2 Low Byte | 0CCH | 0 0 0 0 0 0 0 0 |
| + RCAP2H | T/C 2 Capture Reg. High Byte | 0CBH | 0 0 0 0 0 0 0 0 |
| + RCAP2L | T/C 2 Capture Reg. Low Byte | 0CAH | 0 0 0 0 0 0 0 0 |
| * SCON | Serial Control | 98H | 0 0 0 0 0 0 0 0 |
| SBUF | Serial Data Buffer | 99H | Interminate |
| PCON | Power Control | 87H | HMOS 0 X X X X X X X CHMOS 0 X X X 0 0 0 0 |

Table 7.3 List of all SFRs (* = Bit addressable, + = 8052 only)

| Bit addressable | | 8 Bytes | | | | | | |
|-----------------|-------|---------|--------|--------|-----|-----|------|----|
| F8 | | | | | | | | FF |
| F0 | B | | | | | | | F7 |
| E8 | | | | | | | | EF |
| E0 | ACC | | | | | | | E7 |
| D8 | | | | | | | | DF |
| D0 | PSW | | | | | | | D7 |
| C8 | T2CON | | RCAP2L | RCAP2H | TL2 | TH2 | | CF |
| C0 | | | | | | | | C7 |
| B8 | IP | | | | | | | BF |
| B0 | P3 | | | | | | | B7 |
| A8 | IE | | | | | | | AF |
| A0 | P2 | | | | | | | A7 |
| 98 | SCON | SBUF | | | | | | 9F |
| 90 | P1 | | | | | | | 97 |
| 88 | TCON | TMOD | TL0 | TL1 | TH0 | TH1 | | 8F |
| 80 | P0 | SP | DPL | DPH | | | PCON | 87 |

Table 7.4 SFR memory map

7.4 Memory Organization in 8051

Fig. 7.8 shows the basic memory structure for 8051. It can access upto 64 K program memory and 64 K data memory. The 8051 has 4 Kbytes of internal program memory and 276 bytes of internal data memory.

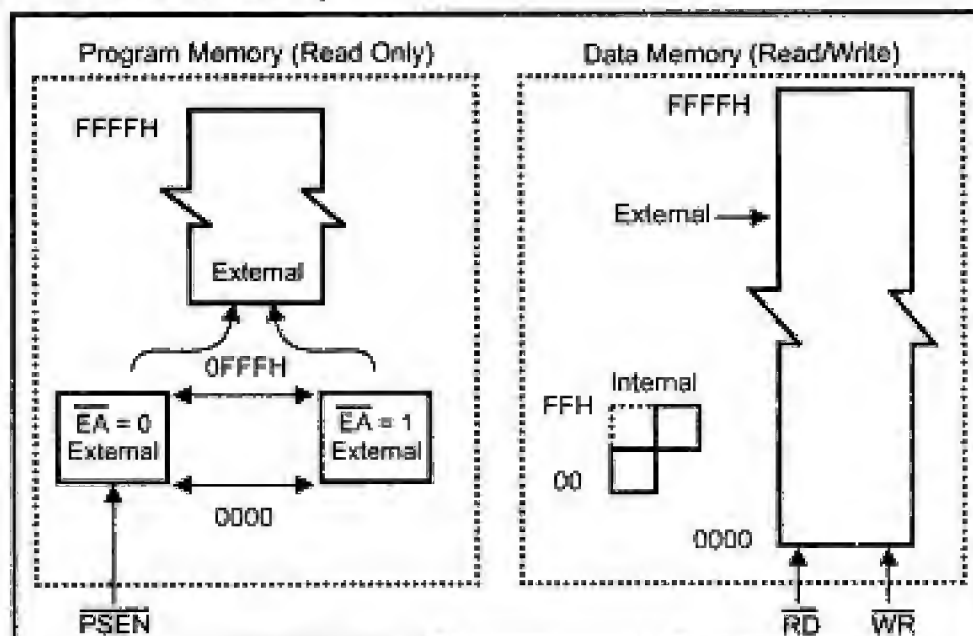


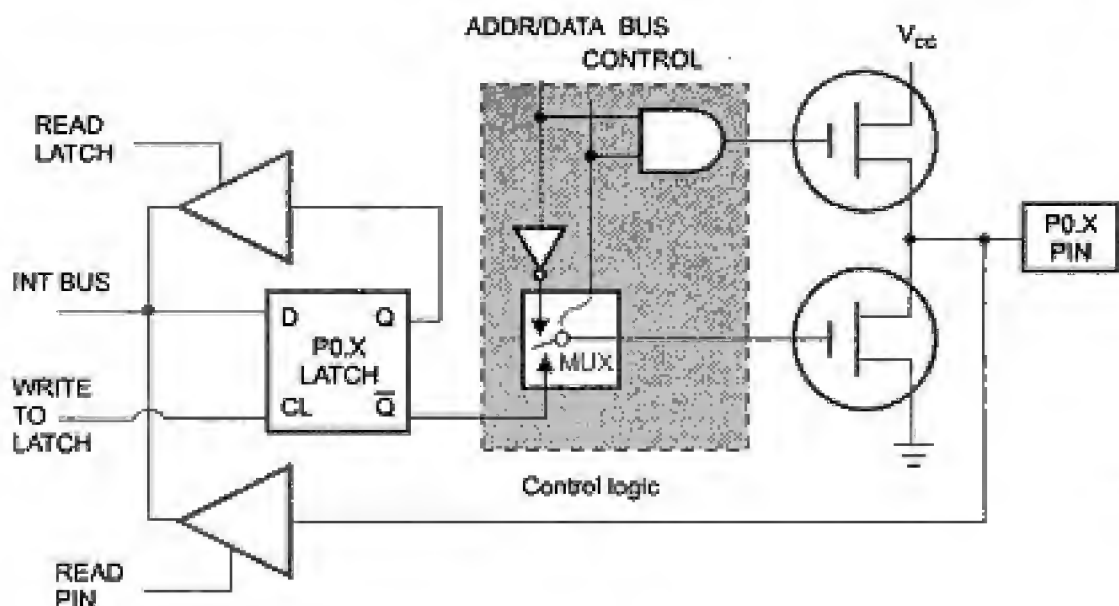
Fig. 7.8 Memory structure

Each memory type has different addressing mechanism, different control signals, and a different function. Each may be added independently, and each uses the same address and data buses, but with different control signals.

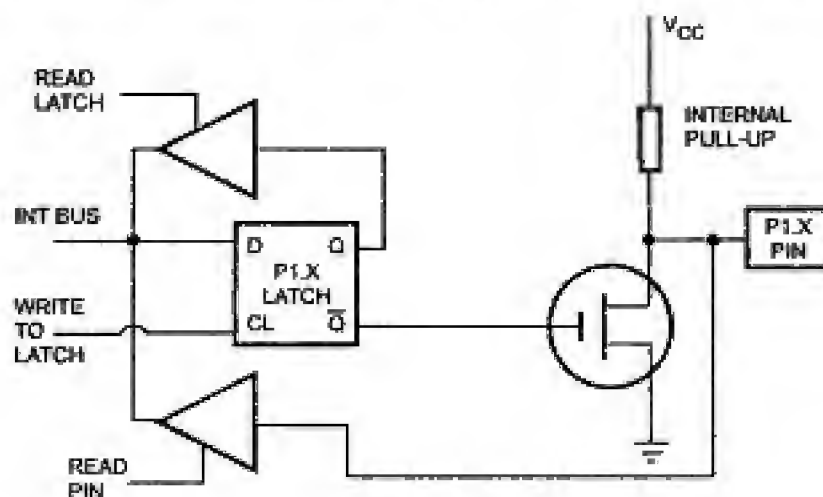
In addition to the program and data memory, there is another physical address space, some of these - B, SP, PSW, DPH and DPL registers are discussed earlier. Others-I/O ports and peripheral function registers collectively referred as special function registers will be introduced in the following sections.

7.5 Input/Output Pins, Ports and Circuits

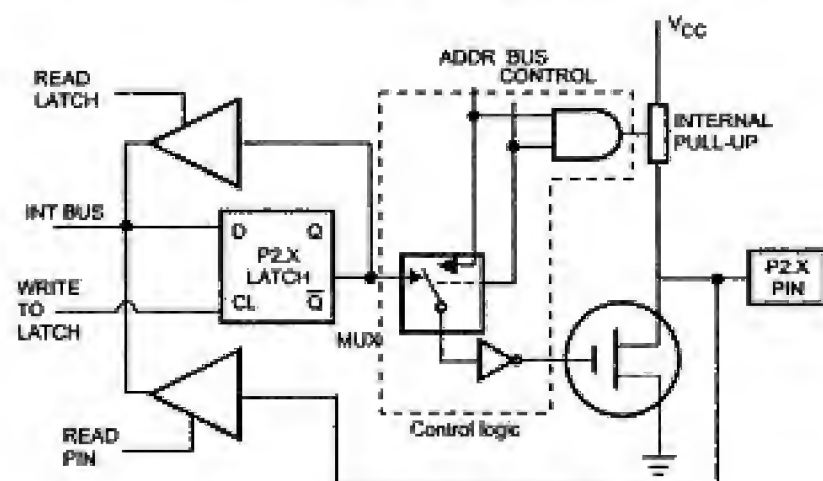
The 8051 provides four port for I/O interfacing. Fig. 7.9 shows a functional diagram of a typical bit latch and I/O buffers in each of the four ports. The bit latch is nothing but the one bit in the port's SFR. It is represented as a D flip-flop. The signal "write to latch" acts as clock input for D flip-flop. The data from the internal bus is clock-in in response to a "write to latch" signal from the CPU. The \bar{Q} output or Q output after inversion from D flip-flop is connected at the gate input of the driver FET. The ON and OFF state of the driver FET due to the data available at the output of latch decides the status of the output pin. It is possible to read Q output of latch by activating "read latch" signal from the CPU. The actual port status can be read by activating "read pin" signal, as shown in Fig. 7.9.



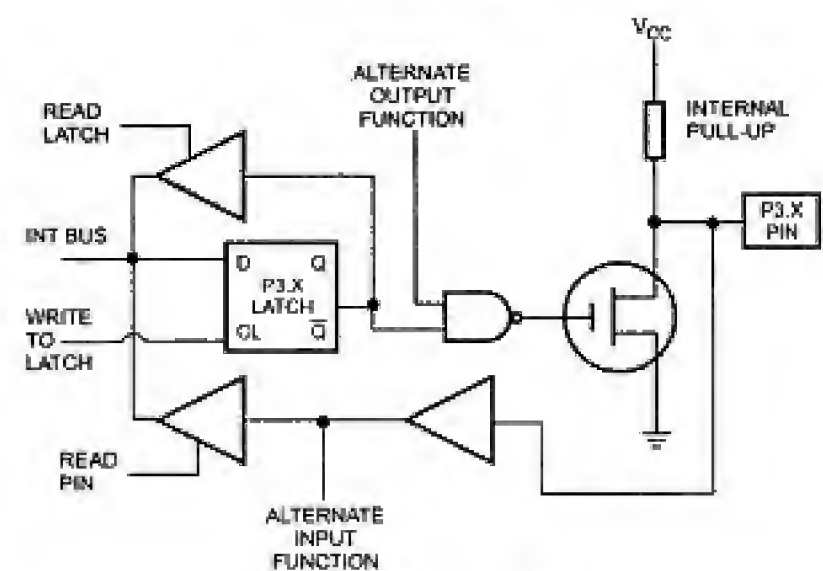
(a) Port 0 bit



(b) Port 1 bit



(c) Port 2 bit



(d) Port 3 bit

Fig. 7.9 8051 port bit latches and I/O buffers

As shown in the Fig. 7.9, for Port 0 and Port 2 drivers are switchable to internal ADDR/DATA and ADDR bus, respectively, by internal CONTROL signal. The switching is required to access external memory. During external memory accesses, the P2 SFR remains unchanged, but P0 SFR gets 1s written to it.

As mentioned earlier, Port 3 has multifunction pins. Therefore, each pin of Port 3 can be programmed to use as I/O or as one of the alternate function. This is achieved by the another control input, "Alternate output function", as shown in the Fig. 7.9. When latch bit of Port 3 contains 1, the output level is controlled by control input, "alternate output function."

The port pin can be configured as an input by writing 1 in the latch bit of the corresponding pin. It turn OFF the output driver FET. Then for, Ports 1,2 and 3, the pin is pulled high by the internal pull-up, but can be pulled low by an external source. There is no internal pull-up for port 0. Therefore, its output pin floats when 1 is written in the latch bit, and pin can be used as a high impedance input. The Port 0 is said to be "true bidirectional", because when configured as an input it floats.

On the otherhand, the output of Ports 1,2 and 3 are pulled high with pull-up registers, when configured as an input. Thus they are sometimes called "quasi bidirectional" ports.

The Table 7.7 summarizes the functions of four ports.

| Port | Functions | | | | | | | | | | | | | | | | |
|-----------|---|----------|-------------------|----------|--------------------|-----------|----------------------|-----------|----------------------|---------|------------------------|---------|------------------------|---------|------------------------------|---------|-----------------------------|
| Port 0 | <ul style="list-style-type: none"> • Used as an input/output port • Used as a bidirectional low-order address and data bus for external memory. | | | | | | | | | | | | | | | | |
| Port 1 | <ul style="list-style-type: none"> • Used as an input/output port. | | | | | | | | | | | | | | | | |
| Port 2 | <ul style="list-style-type: none"> • Used as an input/output port. • Used as a higher-order address bus for external memory. | | | | | | | | | | | | | | | | |
| Port 3 | <ul style="list-style-type: none"> • PinAltem used as an input/output portate use <table> <tr> <td>P3.0-RXD</td><td>Serial data input</td></tr> <tr> <td>P3.1-TXD</td><td>Serial data output</td></tr> <tr> <td>P3.2-INT0</td><td>External interrupt 0</td></tr> <tr> <td>P3.3-INT1</td><td>External interrupt 1</td></tr> <tr> <td>P3.4-T0</td><td>External timer 0 input</td></tr> <tr> <td>P3.7-T1</td><td>External timer 1 input</td></tr> <tr> <td>P3.6-WR</td><td>External memory write signal</td></tr> <tr> <td>P3.7-RD</td><td>External memory read signal</td></tr> </table> | P3.0-RXD | Serial data input | P3.1-TXD | Serial data output | P3.2-INT0 | External interrupt 0 | P3.3-INT1 | External interrupt 1 | P3.4-T0 | External timer 0 input | P3.7-T1 | External timer 1 input | P3.6-WR | External memory write signal | P3.7-RD | External memory read signal |
| P3.0-RXD | Serial data input | | | | | | | | | | | | | | | | |
| P3.1-TXD | Serial data output | | | | | | | | | | | | | | | | |
| P3.2-INT0 | External interrupt 0 | | | | | | | | | | | | | | | | |
| P3.3-INT1 | External interrupt 1 | | | | | | | | | | | | | | | | |
| P3.4-T0 | External timer 0 input | | | | | | | | | | | | | | | | |
| P3.7-T1 | External timer 1 input | | | | | | | | | | | | | | | | |
| P3.6-WR | External memory write signal | | | | | | | | | | | | | | | | |
| P3.7-RD | External memory read signal | | | | | | | | | | | | | | | | |

Table 7.5 Port functions

7.6 Timers and Counters

The 8051 has two 16-bit Timer/Counter registers : Timer 0 and Timer 1. Both can be configured to operate either as timers or event counters.

In the "Timer" mode, the register is incremented after every machine cycle. Thus, one can think of it as counting machine cycles. Since a machine cycle consists of 12 oscillator periods, the count rate is $\frac{1}{12}$ of the oscillator frequency. For example, if crystal frequency is 12 MHz, then the timer clock frequency is 1 MHz.

In the "Counter" mode, the register is incremented in response to a 1-to-0 transition at its corresponding external input pin, T_0 or T_1 . In this mode, the external input is sampled during phase 2 of state 5 of every machine cycle. When the samples show a high in one cycle and a low in the next cycle, the count is incremented. The new count value appears in the register during phase 1 of state 3 of the cycle following the one in which the transition was detected. Since it takes 2 machine cycles (24 oscillator periods) to recognize a 1-to-0 transition, the maximum count rate is $\frac{1}{24}$ of the oscillator frequency. There are no restrictions on the duty cycle of the external input signal, but to ensure that a given level is sampled atleast once before it changes, it should be held for atleast one full machine cycle.

7.6.1 Timer/Counter Control Logic

The timers and counters of 8051 are controlled by TR1/0 bits in TCON, gate bits in TMOD and $\overline{INT1/0}$ input pins of 8051. The C/ \overline{T} bit in the TMOD register decides the operation - bit is 0 for timer selection and bit is 1 for counter operation. This is illustrated in Fig. 7.10

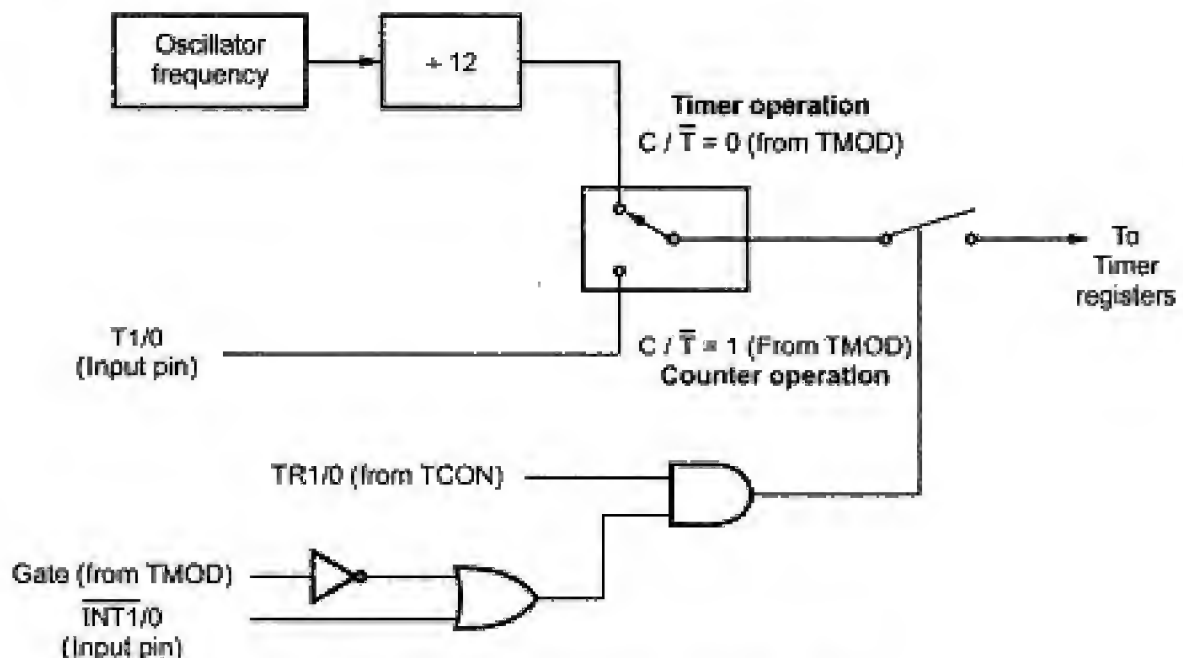


Fig. 7.10 Timer/counter control logic

7.6.2 Timer 0 and Timer 1

In this timers, "Timer" or Counter" mode is selected by control bits C/\bar{T} in the Special Function Register TMOD (Fig. 7.11). These two Timer/Counters have four operating modes, which are selected by bit-pairs (M1, M0) in TMOD. Modes 0, 1 and 2 are same for both Timer/Counters. Mode 3 is different. The four operating modes are described as follows :

| (MSB) | | | | (LSB) | | | |
|-------------|--|-------------|--|---------|-------------|---|----|
| GATE | C/\bar{T} | M1 | M0 | GATE | C/\bar{T} | M1 | M0 |
| Timer 1 | | | | Timer 0 | | | |
| GATE | Gating control when set. Timer/Counter "x" is enabled only while "INTx" pin is high and "TRx" control bit is set. When cleared Timer "x" is enabled whenever "TRx" control bit is set. | C/\bar{T} | Timer or Counter selector cleared for Timer operation (input from internal system clock). Set for Counter operation (input from "Tx" input pin). | M1 | M0 | Operating Mode | |
| | | | | 0 | 0 | 8-bit Timer/Counter "THx" with "TLx" 5-bit prescaler. | |
| C/\bar{T} | | | | 0 | 1 | 16-bit Timer/Counter "THx" with "TLx" are cascaded; there is no prescaler. | |
| | | | | 1 | 0 | 8-bit auto-reload Timer/Counter "THx" holds a value which is to be reloaded into "TLx" each time it overflows. | |
| | | | | 1 | 1 | (Timer 0) TL0 is an 8-bit Timer/Counter controlled by the standard Timer 0 control bits. TH0 is an 8-bit timer only controlled by Timer 1 control bits. | |
| | | | | 1 | 1 | (Timer 1) Timer/Counter 1 stopped. | |

Fig. 7.11 TMOD : Timer/counter mode control register

MODE 0

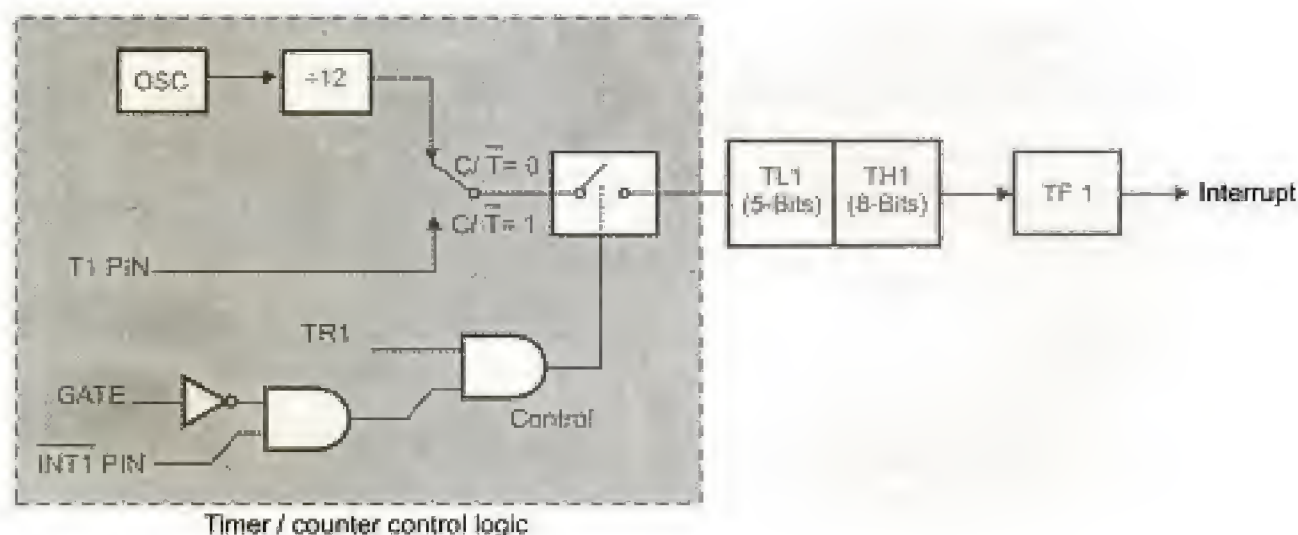


Fig. 7.12 Timer/counter 1 mode 0 : 13-bit counter

Both Timers in Mode 0 is an 8-bit Counter with a divide-by-32 prescaler. This 13-bit timer is MCS-48 compatible. Fig. 7.12 shows the Mode 0 operation as it applies to Timer 1. In this mode, the Timer register is configured as a 13-bit register. As the count rolls over from all 1s to all 0s, it sets the Timer interrupt flag $TF1$. The counted input is enabled to the Timer when $TR1 = 1$ and either $GATE = 0$ or $\overline{INT1} = 1$. (Setting $GATE = 1$ allows the Timer to be controlled by external input $\overline{INT1}$, to facilitate pulse width measurements.) $TR1$ is a control bit in the Special Function Register TCON (Fig. 7.13) $GATE$ is in TMOD.

| (MSB) | | | | (LSB) | | | |
|-------|-----|-----|-----|-------|-----|-----|-----|
| TF1 | TR1 | TF0 | TR0 | IE1 | IT1 | IE0 | IT0 |

| Symbol | Position | Name and significance |
|--------|----------|--|
| TF1 | TCON.7 | Timer 1 Overflow Flag. Set by hardware on timer/counter overflow. Cleared when interrupt processed. |
| TR1 | TCON.6 | Timer 1 Run control bit. Set/cleared by software to turn timer/counter on/off. |
| TF0 | TCON.5 | Timer 0 Overflow Flag. Set by hardware on timer/counter overflow. Cleared when interrupt processed. |
| TR0 | TCON.4 | Timer 0 Run control bit. Set/cleared by software to turn timer/counter on/off. |
| IE1 | TCON.3 | Interrupt 1 Edge Flag. Set by hardware when external interrupt edge detected. Cleared when interrupt processed. |
| IT1 | TCON.2 | Interrupt 1 Type control bit. Set/cleared by software to specify falling edge/low level triggered external interrupts. |
| IE0 | TCON.1 | Interrupt 0 Edge Flag. Set by hardware when external interrupt edge detected. Cleared when interrupt processed. |
| IT0 | TCON.0 | Interrupt 0 Type control bit. Set/cleared by software to specify falling edge/low level triggered external interrupts. |

Fig. 7.13 TCON-timer/counter control/status register

The 13-bit register consists of all 8-bits of TH1 and the lower 5-bits of TL1. The upper 3-bits of TL1 are indeterminate and should be ignored. Setting the run flag ($TR1$) does not clear the registers.

Mode 0 operation is the same for Timer 0 as for Timer 1. Substitute $TR0$, $TF0$ and $\overline{INT0}$ for the corresponding Timer 1 signals in Fig. 7.12. There are two different $GATE$ bits, one for Timer 1 (TMOD.7) and one for Timer 0 (TMOD.3).

MODE 1

Mode 1 is the same as Mode 0, except that the Timer register is being run with all 16-bits.

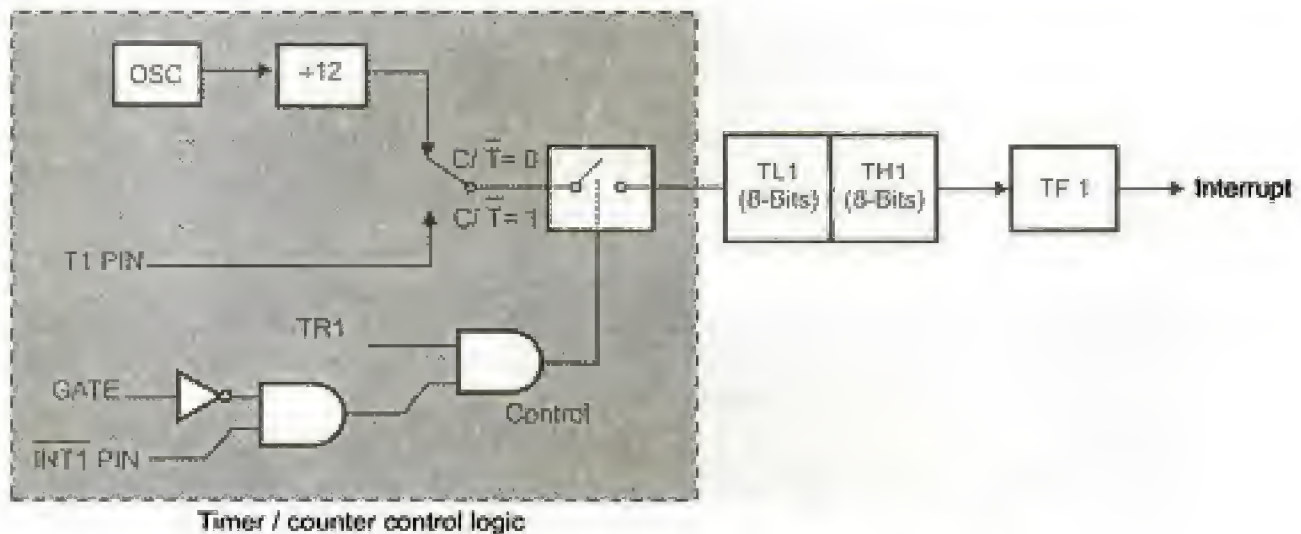


Fig. 7.14 Timer/counter 1 mode 0 : 13-bit counter

MODE 2

Mode 2 configures the Timer register as an 8-bit Counter (TL1) with automatic reload, as shown in Fig. 7.15. Overflow from TL1 not only sets TF1, but also reloads TL1 with the contents of TH1, which is preset by software. The reload leaves TH1 unchanged. Mode 2 operation is the same for Timer/Counter 0.

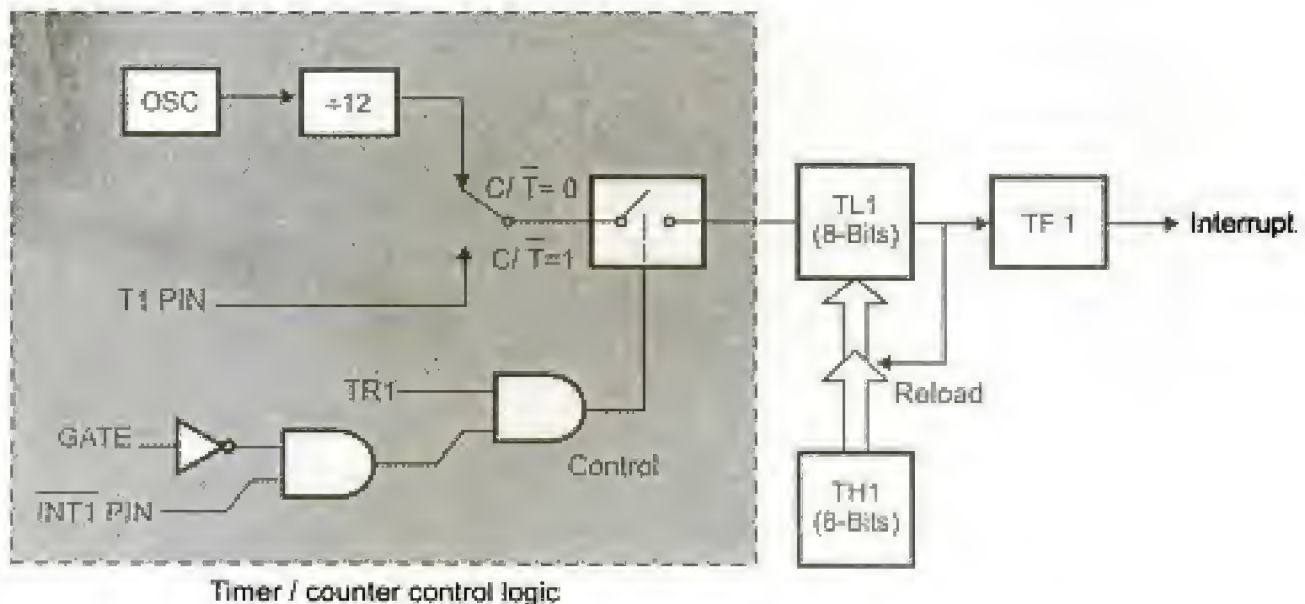


Fig. 7.15 Timer/counter 1 mode 2 : 8-bit auto reload

MODE 3

Timer 1 in Mode 3 simply holds its count. The effect is the same as setting $TR1 = 0$. Timer 0 in Mode 3 establishes TL0 and TH0 as two separate counters. The logic for Mode 3 on Timer 0 is shown in Fig. 7.16. TL0 uses the Timer 0 control bits : C/\bar{T} , GATE, $TR0$, $\overline{INT0}$, and $TF0$. TH0 is locked into a timer mode (counting machine cycles) and takes over the use of $TR1$ and $TF1$ from Timer 1. Thus, TH0 now controls the : Timer 1 interrupt.

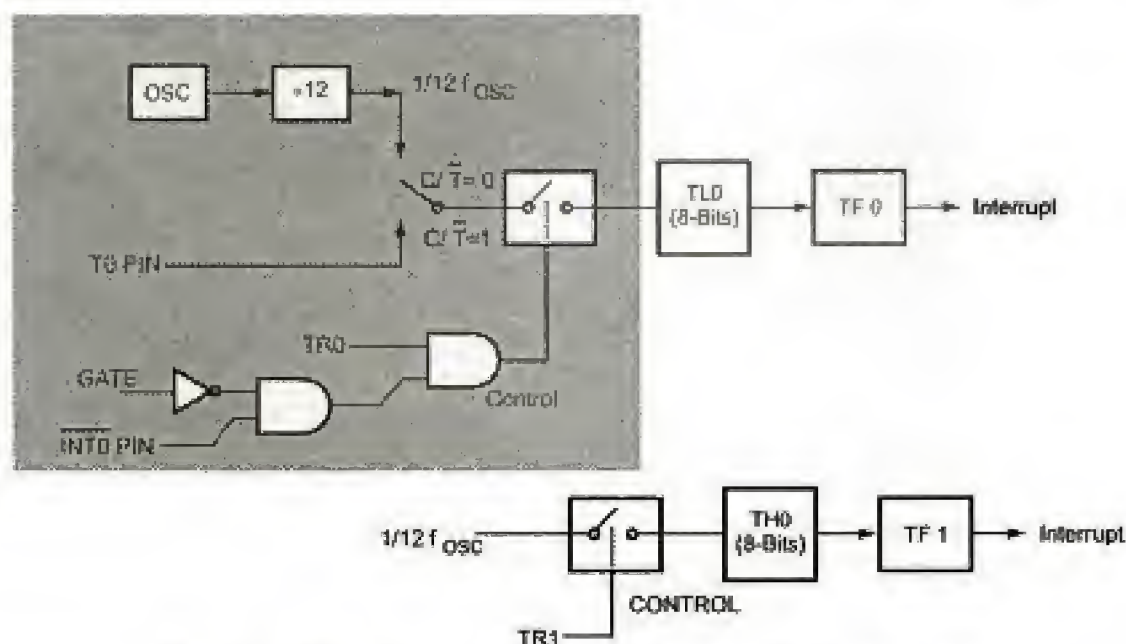


Fig. 7.16 Timer/counter 0 mode 3 : two 8-bit counters

Mode 3 is provided for applications requiring an extra 8-bit timer or counter. With Timer 0 in Mode 3, an 8051 can look like it has three Timer/Counters, and an 8052, like it has four. When timer 0 is in Mode 3, Timer 1 can be turned on and off by switching it out of and into its own Mode 3, or can still be used by the serial port as a baud rate generator, or in fact, in any application not requiring an interrupt

The Table 7.6 summarizes the modes of timers.

| Mode | Brief description |
|--------|--|
| Mode 0 | 13-bit timer (TL-5bits and TH-8bits). Counter overflow is indicated by time interrupt flag. |
| Mode 1 | 16-bit timer (TL-8bits and TH-8bits) Reset is same as mode 0. |
| Mode 2 | Automatic reload mode. 8-bit counter (TL-8bit) overflow from TL not only sets TF, but also reloads TL with the contents of TH. |
| Mode 3 | Establishes TL and TH as two separate counters. |

Table 7.6 Summary of timer modes

7.7 Serial Port

The serial port of 8051 is full duplex, means it can transmit and receive simultaneously. It uses register SBUF to hold data. Register SCON controls data communication, register-PCON controls data rates, and pin RXD (P3.0) and TXD (P3.1) do the data transfer. The serial port receive and transmit registers are both accessed at special function register SBUF. Writing to SBUF loads the transmit register, and reading SBUF accesses a physically separate receive register. Both mutually exclusive registers use address 99H. The serial port of 8051 is receive-buffered, meaning it can commence reception of a second byte before a previously received byte has been read from the receive register. (However, if the first byte still hasn't been read by the time reception of the second byte is complete, one of the bytes will be lost).

The 8051 provides four programmable modes for serial data communication. A particular mode can be selected by setting the SM0 and SM1 bits in SCON. The mode selection also decides the baud rate. The Fig. 7.17 (a) and (b) shows the bit patterns for SCON and PCON.

| (MSB) | | | | (LSB) | | | |
|-------|-----|-----|-----|-------|-----|----|----|
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| SM0 | SM1 | SM2 | REN | TB8 | RB8 | TI | RI |

| Symbol | Position | Name and significance |
|--------|----------|---|
| SM0 | SCON.7 | Serial port Mode control bit-0. Set/cleared by software (see note). |
| SM1 | SCON.6 | Serial port Mode control bit-1. Set/cleared by software (see note). |
| SM2 | SCON.5 | Serial port Mode control bit-2. Set by software to disable reception of frames for which bit 8 is zero. |
| REN | SCON.4 | Receiver Enable control bit. Set/cleared by software to enable/disable serial data reception. |
| TB8 | SCON.3 | Transmit bit-8. Set/cleared by hardware to determine state of ninth data bit transmitted in 9-bit UART mode. |
| RB8 | SCON.2 | Receive bit-8. Set/cleared by hardware to indicate state of ninth data bit received. |
| TI | SCON.1 | Transmit Interrupt flag. Set by hardware when byte transmitted. Cleared by software after servicing. |

| | | |
|----|------------------------|---|
| RI | SCON.0 | Receive Interrupt flag. Set by hardware when byte received. Cleared by software after servicing. |
| | Note : Mode | The state of (SM0, SM1) selects : |
| | 0 | 0 0 - Shift register ; baud = $f/12$ |
| | 1 | 0 1 - 8-bit UART, variable data rate. |
| | 2 | 1 0 - 9-bit UART, fixed data rate ; baud = $f/32$ or $f/64$ |
| | 3 | 1 1 - 9-bit UART, variable data rate. |

Fig. 7.17 (a) SCON-serial port control/status register

| | | | | | | | | |
|-------|------|---|---|---|-----|-----|----|-------|
| (MSB) | 7 | 6 | 5 | 4 | 3 | 2 | 1 | (LSB) |
| | SMOD | — | — | — | GF1 | GF0 | PD | IDL |

| Symbol | Position | Name and significance |
|---|----------|---|
| SMOD | PCON.7 | Serial baud rate modify bit. It is 0 at reset. It is set to 1 by program to double the baud rate. |
| — | PCON.6-4 | Not defined |
| GF1 | PCON.3 | General purpose user flag bit-1. Set/cleared by program. |
| GF0 | PCON.2 | General purpose user flag bit-0. Set/cleared by program. |
| PD | PCON.1 | Power down bit. It is set to 1 by program to enter power down configuration for CHMOS microcontrollers. |
| IDL | PCON.0 | Idle mode bit. It is set to 1 by program to enter idle mode configuration for CHMOS microcontrollers. |
| Note : PCON is not bit addressable | | |

Fig. 7.17 (b) PCON register

7.7.1 Operating Modes for Serial Port

MODE 0

In this mode, serial data enters and exits through RXD. TXD outputs the shift clock. 8 bits are transmitted/received : 8 data bits (LSB first). The baud rate is fixed at $1/12$ the oscillator frequency.

MODE 1

In this mode, 10-bits are transmitted (through TXD) or received (through RXD) : a start bit (0), 8 data bits (LSB first), and a stop bit (1). On receive, the stop bit goes into RB8 in Special Function Register SCON. The baud rate is variable.

MODE 2

In this mode, 11-bits are transmitted (through TXD) or received (through RXD) : a start bit (0), 8 data bits (LSB first), a programmable 9th data bit, and a stop bit (1). On Transmit, the 9th data bit (TB8 in SCON) can be assigned the value of 0 or 1. Or, for example, the parity bit (P, in the PSW) could be moved into TB8. On receive, the 9th data bit goes into RB8 in Special Function Register SCON, while the stop bit is ignored. The baud rate is programmable to either $\frac{1}{32}$ or $\frac{1}{64}$ the oscillator frequency.

MODE 3

In this mode, 11-bits are transmitted (through TXD) or received (through RXD) : a start bit (0), 8 data bits (LSB first), a programmable 9th data bit and a stop bit (1). In fact, Mode 3 is the same as Mode 2 in all respects except the baud rate. The baud rate in Mode 3 is variable.

In all four modes, transmission is initiated by any instruction that uses SBUF as a destination register. Reception is initiated in Mode 0 by the condition RI = 0 and REN = 1. Reception is initiated in the other modes by the incoming start bit if REN = 1.

The Table 7.7 summaries the four serial port modes provided by 8051.

| Mode | Transmission format | Baud rate |
|------|---|--|
| 0 | 8 data bits | $\frac{1}{12}$ oscillator frequency |
| 1 | 10-bit (start bit + 8 data bits + stop bit) | Variable |
| 2 | 11-bit (start bit + 8 data bits + programmable 9 th data bit + stop bit) | Programmable to either $\frac{1}{32}$ or $\frac{1}{64}$ oscillator frequency |
| 3 | 11-bit (start bit + 8 data bit + programmable 9 th data bit + stop bit) | Variable |

Table 7.7 Summary of serial port modes

7.7.2 Serial Port Control Register

The serial port control and status register is the Special Function Register SCON, shown in Fig. 7.18. This register contains not only the mode selection bits, but also the 9th data bit for transmit and receive (TB8 and RB8), and the serial port interrupt bits (TI and RI).

7.7.3 Generating Baud Rates**Serial Port in Mode 0 :**

Mode 0 has a fixed baud rate which is $\frac{1}{12}$ of the oscillator frequency. To run the serial port in this mode none of the Timer/Counters need to be set up. Only the SCON register needs to be defined.

$$\text{Baud rate} = \frac{\text{Osc. freq.}}{12}$$

Serial Port in Mode 1

Mode 1 has a variable baud rate. The baud rate can be generated by either Timer 1 or Timer 2 (8052 only).

Using Timer/Counter 1 to Generate Baud Rates

For this purpose, Timer 1 is used in mode 2 (Auto-Reload).

$$\text{Baud rate} = \frac{k \times \text{Oscillator freq.}}{32 \times 12 \times [256 - \text{TH1}]}$$

If SMOD = 0, then $k = 1$.

If SMOD = 1, then $k = 2$. (SMOD is the PCON register)

Most of the time the user knows the baud rate and needs to know the reload value for TH1. Therefore, the equation to calculate TH1 can be written as :

$$\text{TH1} = 256 - \frac{k \times \text{Osc. freq.}}{384 \times \text{Baud rate}}$$

TH1 must be an integer value. Rounding off TH1 to the nearest integer may not produce the desired baud rate. In this case, the user may have to choose another crystal frequency.

Since the PCON register is not bit addressable, one way to set the bit is logical ORing the PCON register. (i.e. ORL PCON, #80H). The address of PCON is 87H.

Using Timer/Counter 2 to Generate Baud Rates

For this purpose, Timer 2 must be used in the baud rate generating mode. If Timer 2 is being clocked through pin T2 (P1.0) the baud rate is :

$$\text{Baud rate} = \frac{\text{Timer 2 overflow rate}}{16}$$

And if it is being clocked internally the baud rate is :

$$\text{Baud rate} = \frac{\text{Osc. freq.}}{32 \times [65536 - (\text{RCAP2H}, \text{RCAP2L})]}$$

To obtain the reload value for RCAP2H and RECAP2L the above equation can be rewritten as :

$$\text{RCAP2H}, \text{RCAP2L} = 65536 - \frac{\text{Osc. freq.}}{32 \times \text{Baud rate}}$$

Serial Port in Mode 2

The baud rate is fixed in this mode and is $\frac{1}{32}$ or $\frac{1}{64}$ of the oscillator frequency depending on the value of the SMOD bit in the PCON register. In this mode none of the Timers are used and the clock comes from the internal phase 2 clock.

SMOD = 1, Baud rate = $\frac{1}{32}$ Osc. freq.

SMOD = 0, Baud rate = $\frac{1}{64}$ Osc. freq.

To set the SMOD bit : ORL PCON,#80H. The address of PCON is 87H.

Serial Port in Mode 3

The baud rate in mode 3 is variable and sets up exactly the same as in mode 1.

7.8 Interrupt Structure

The 8051 provides 5 interrupt sources. The 8052 provides 6. These are shown in Fig. 7.18. The external Interrupts $\overline{\text{INT0}}$ and $\overline{\text{INT1}}$ can each be either level-activated or transition-activated, depending on bits IT0 and IT1 in Register TCON. The flags that actually generate these interrupts are bits IE0 and IE1 in TCON. When an external interrupt is generated, the flag that generated it is cleared by the hardware when the service routine is vectored to only if the interrupt was transition-activated. If the interrupt was level-activated, then the external requesting source is what controls the request flag, rather than the on-chip hardware.

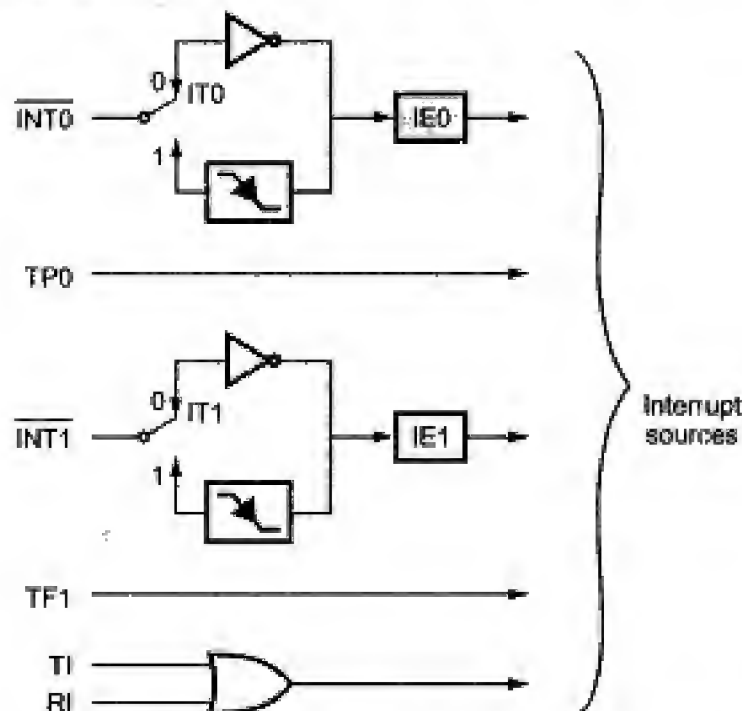


Fig. 7.18 MCS - 51 Interrupt structure

The Timer 0 and Timer 1 Interrupts are generated by TF0 and TF1, which are set by a rollover in their respective Timer/Counter registers (except see Timer 0 in Mode 3). The timer flag set upon generation of interrupt is cleared by the on-chip hardware when microcontroller starts execution of particular interrupt service routine.

The Serial port Interrupt is generated by the logical OR of RI and TI. Neither of these flags is cleared by hardware when the service routine is vectored to. In fact, the service

routine will normally have to determine whether it was RI or TI that generated the interrupt, and the bit will have to be cleared in software.

All of the bits that generate interrupts can be set or cleared by software, with the same result as though it had been set or cleared by hardware. That is, interrupts can be generated or pending interrupts can be cancelled in software.

Each of these interrupt sources can be individually enabled or disabled by setting or clearing a bit in special Function Register IE (Fig. 7.19). IE contains also a global disable bit, EA, which disables all interrupts at once.

Note in Fig. 7.19 that bit position IE.6 is unimplemented. In the 8051s, bit position IE.5 is also unimplemented. User software should not write 1s to these bit positions, since they may be used in future MCS-51 products.

| (MSB) | | | | (LSB) | | | |
|-------|---|-----|----|-------|-----|-----|-----|
| EA | - | ET2 | ES | ET1 | EX1 | ET0 | EX0 |

| Symbol | Position | Name and significance |
|--------|----------|--|
| EA | IE.7 | Enable All control bit. Cleared by software to disable all interrupts, independent of the state of IE.4-IE.0. |
| - | IE.6 | (Reserved) |
| ET2 | IE.5 | (Reserved) |
| ES | IE.4 | Enable Serial port control bit. Set/cleared by software to enable/disable interrupts from TI or RI flags. |
| ET1 | IE.3 | Enable Timer 1 control bit. Set/cleared by software to enable/disable interrupts from timer/counter 1 |
| EX1 | IE.2 | Enable External interrupt 1 control bit. Set/cleared by software to enable/ disable interrupts from INT1. |
| ET0 | IE.1 | Enable Timer 0 control bit. Set/cleared by software to enable/disable interrupts from timer/counter 0. |
| EX0 | IE.0 | Enable external interrupt 0 control bit. Set/cleared by software to enable/disable interrupts from INTO. |

Fig. 7.19 IE-Interrupt enable register

7.8.1 Priority Level Structure

Each interrupt source can also be individually programmed to one of two priority levels by setting or clearing a bit in Special Function Register IP (Fig. 7.20). A low-priority interrupt can itself be interrupted by a high-priority interrupt, but not by another low priority interrupt. A high-priority interrupt can't be interrupted by any other interrupt source.

| (MSB) | | | | (LSB) | | | |
|-------|---|---|----|-------|-----|-----|-----|
| - | - | - | PS | PT1 | PX1 | PT0 | PX0 |

| Symbol | Position | Name and Significance |
|--------|----------|---|
| - | IP.7 | (Reserved) |
| - | IP.6 | (Reserved) |
| - | IP.5 | (Reserved) |
| PS | IP.4 | Serial port Priority control bit. Set/cleared by software to specify high/low priority interrupts for Serial port. |
| PT1 | IP.3 | Timer 1 Priority control bit. Set/cleared by software to specify high/low priority interrupts for timer/counter 1. |
| PX1 | IP.2 | External interrupt 1 Priority control bit. Set/cleared by software to specify high/low priority interrupts for INT1. |
| PT0 | IP.1 | Timer 0 Priority control bit. Set/cleared by software to specify high/low priority interrupts for timer/counter 0. |
| PX0 | IP.0 | External interrupt 0 Priority control bit. Set/cleared by software to specify high/low priority interrupts for INT0. |

Fig. 7.20 IP - Interrupt priority control register

If two requests of different priority levels are received simultaneously, the request of higher priority level is serviced. If requests of the same priority level are received simultaneously, an internal polling sequence determines which request is serviced. Thus within each priority level there is a second priority structure determined by the polling sequence, as follows :

| No. | Source | Priority within Level |
|-----|---------|-----------------------|
| 1. | IE0 | (highest) |
| 2. | TF0 | |
| 3. | IE1 | |
| 4. | TF1 | |
| 5. | RI + TI | (lowest) |

Note that the "priority within level" structure is only used to resolve simultaneous requests of the same priority level.

The IP register contains a number of unimplemented bits. IP.7 and IP.6 are vacant in the 8052s, and in the 8051s these and IP.5 are vacant. User software should not write 1s to these bit positions, since they may be used in future MCS-51 products.

7.8.2 External Interrupts

The external sources can be programmed to be level-activated or transition-activated by setting or clearing bit IT1 or IT0 in Register TCON. If $ITx = 0$, external interrupt x is triggered by a detected low at the \overline{INTx} pin. If $ITx = 1$, external interrupt x is edge-triggered. In this mode if successive samples of the \overline{INTx} pin show a high in one cycle and a low in the next cycle, interrupt request flag IEx in TCON is set. Flag bit IEx then requests the interrupt.

Since the external interrupt pins are sampled once each machine cycle, an input high or low should hold for atleast 12 oscillator periods to ensure sampling. If the external interrupt is transition-activated, the external source has to hold the request pin high for atleast one machine cycle, and then hold it low for atleast one machine cycle to ensure that the transition is seen so that interrupt request flag IEx will be set. IEx will be automatically cleared by the CPU when the service routine is called.

If the external interrupt is level-activated, the external source has to hold the request active until the requested interrupt is actually generated. Then it has to deactivate the request before the interrupt service routine is completed, or else another interrupt will be generated.

7.8.3 Single-Step Operation

The 8051 interrupt structure allows single-step execution with very little software overhead. As previously noted, an interrupt request will not be responded to while an interrupt of equal priority level is still in progress, nor will it be responded to after RETI until atleast one other instruction has been executed. This, once an interrupt routine has been entered, it cannot be re-entered until atleast one instruction of the interrupted program is executed. One way to use this feature for single-step operation is to program one of the external interrupts (say, $\overline{INT0}$) to be level-activated. The service routine for the interrupt will terminate with the following code :

```
JNB P3.2$    ;    Wait Here Till  $\overline{INT0}$  Goes High
JB P3.2$     ;    Now Wait Here Till it Goes Low
RETI         ;    Go back and Execute One Instruction
```

Now if $\overline{INT0}$ pin, which is also the P3.2 pin, is held normally low, the CPU will go right into the External Interrupt 0 routine and stay there until $\overline{INT0}$ is pulsed (from low to high to low). Then it will execute RETI, go back to the task program, execute one instruction, and immediately re-enter the External Interrupt 0 routine to await the next pulsing of P3.2. One step of the task program is executed each time P3.2 is pulsed.

7.9 Clock and Oscillator

A machine cycle consists of a sequence of 6 states numbered S1 to S6 (12 oscillator periods). Each state is divided into two phases : Phase 1 and Phase 2. During Phase 1, Phase 1 clock is active, and a Phase 2 clock is active during Phase 2. Thus, a machine cycle

consists of 12 oscillator periods, numbered S1P1 (State 1, Phase 1), to S6P2 (State 6, Phase 2) or 1 μ sec. If the oscillator clock frequency is 12 MHz. Each phase lasts for one oscillator period. Each state lasts for two oscillator periods. Typically, arithmetic and logical operations take place during Phase 1 and internal register-to-register transfers take place during Phase 2.

The diagrams in Fig 7.21 show the fetch/execute timing referenced to the internal states and phases. Since these internal clock signals are not user accessible, the XTAL2 oscillator signal and the ALE (Address Latch Enable) signal are shown for external reference. ALE is normally activated twice during each machine cycle : once during S1P2 and S2P1, and again during S4P2 and S5P1.

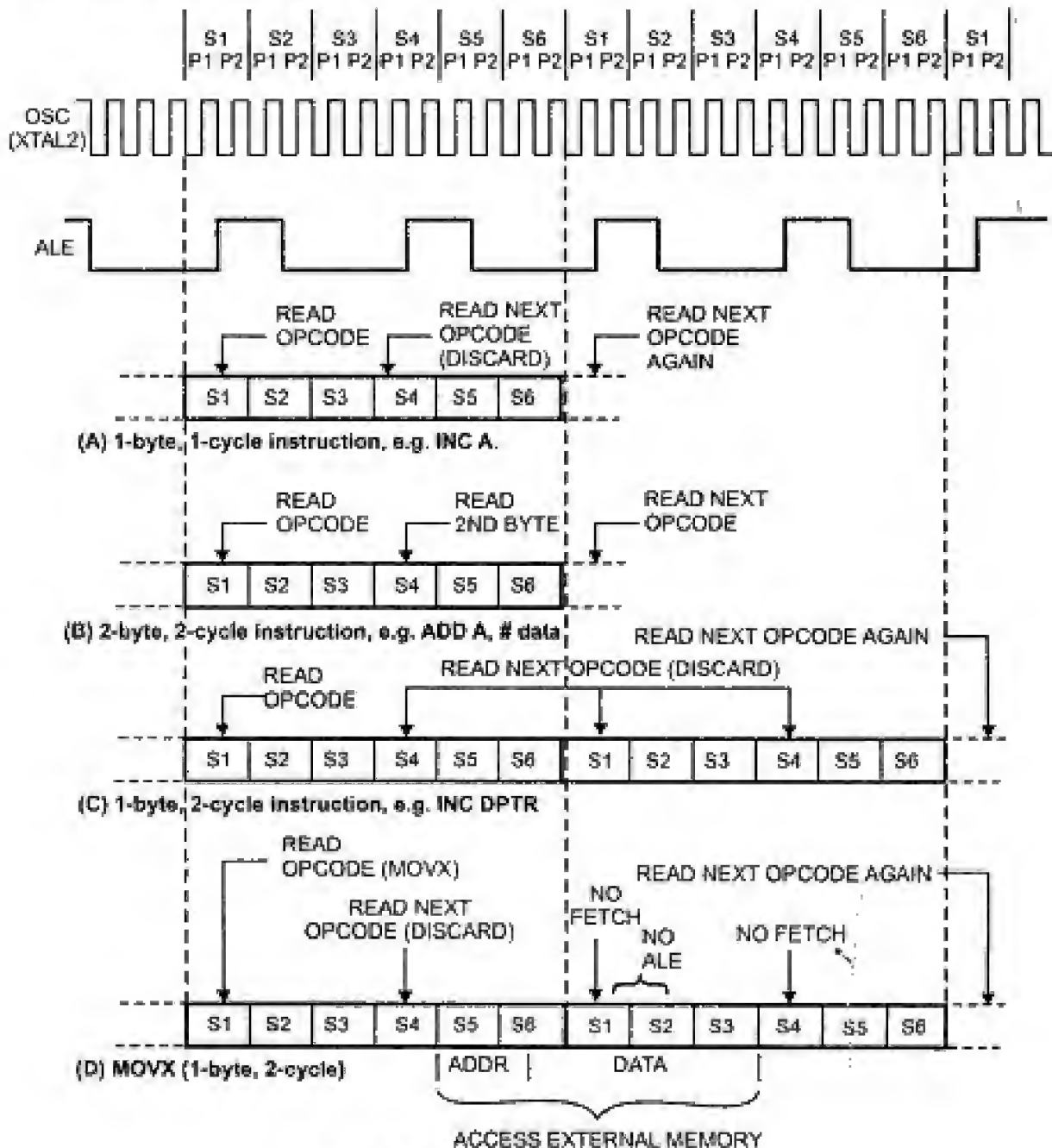


Fig. 7.21 Fetch/Execute sequence

Execution of a one-cycle instruction begins at S1P2, when the opcode is latched into the Instruction Register. If it is a two-byte instruction, the second byte is read during S4 of the same machine cycle. If it is a one-byte instruction, there is still a fetch at S4, but the byte read (which would be the next opcode), is ignored, and the Program Counter is not incremented. In any case, execution is completed at the end of S6P2. Fig. 7.21 (a) and (b) shown the timing for a 1-byte, 1-cycle instruction and for a 2-byte, 1-cycle instruction.

Most 8051 instructions execute in one cycle. MUL (multiply) and DIV (divide) are the only instructions that take more than two cycles to complete. They take four cycles.

Normally, two code bytes are fetched from Program Memory during every machine cycle. The only exception to this is when a MOVX instruction is executed. MOVX is a 1-byte 2-cycle instruction that accesses external Data Memory. During a MOVX, two fetches are skipped while the external Data Memory is being addressed and strobed. Fig. 7.21 (c) and (d) shown the timing for a normal 1-byte, 2-cycle instruction and for a MOVX instruction.

7.10 Power Saving Options

The concept of reduction in power consumption is not just to reduce the electric bill and battery capacities, but due to reduction in power consumption :

- 1) It is able to put more functionality into smaller space.
- 2) It allows the use of smaller and lighter power supplies, and less heat being generated allows denser packaging of circuit components. Expensive fans and blowers can usually eliminated.
- 3) It is important to note that cooler running chip is more reliable, since most random and wearout failures relate to die temperature.

The 8051 has two power saving features. These are the idle and power down modes of operation. The Fig. 7.22 shows the on chip hardware that implements these reduced power modes. Both modes are invoked by software.

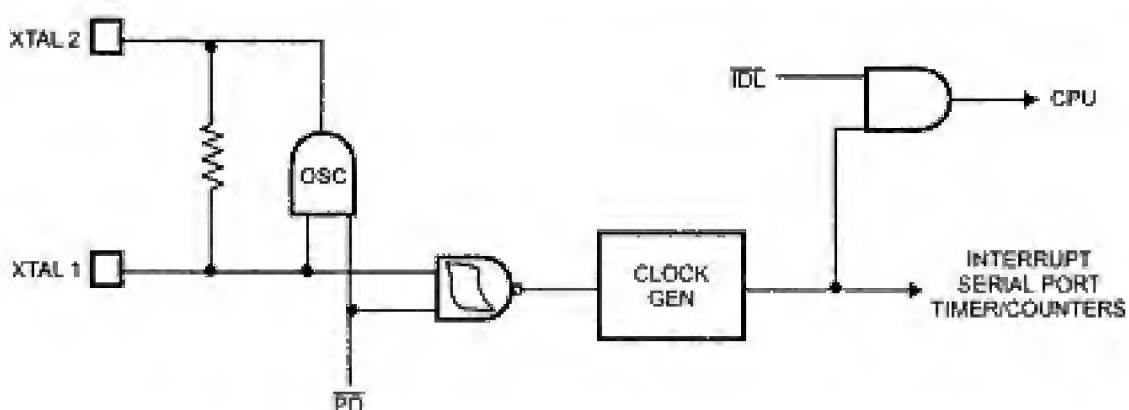


Fig. 7.22 Hardware to implement reduced power modes

7.10.1 Idle Mode

In the Idle Mode ($IDL = 0$ in Fig. 7.22), the CPU puts itself to sleep by gating off its own clock. It doesn't stop the oscillator. It just stop the internal clock signal from getting to the CPU. Since the CPU draws 80 to 90 percent of the chip's power, shutting it off represents a fairly significant power savings. The on-chip peripherals (timers, serial port, interrupts, etc.) and RAM continue to function as normal. The Stack Pointer, Program Counter, Program Status Word, Accumulator and all other registers maintain their data during Idle Mode.

The Idle Mode is invoked by setting bit 0 (IDL) of the PCON register. PCON is not bit-addressable, so the bit has to be set by a byte operation, such as

ORL PCON, # 1

The PCON register also contains flag bits $GF0$ and $GF1$, which can be used for any general purposes, or to give an indication if an interrupt occurred during normal operation or during Idle Mode. In this application, the instruction that invokes Idle Mode also sets one or both of the flag bits. Their status can then be checked in the interrupt routines.

While the device is in the Idle Mode, ALE and \overline{PSEN} are logic high (V_{OH}). Due to this external EPROM can be deselected and have its output disabled.

The port pins hold the logical states they had at the time the Idle was activated. If the device is executing out of external program memory, Port 0 is left in a high impedance state and Port 2 continues to output the high byte of the program counter (using the strong pullups to output 1s.) If the device is executing out of internal program memory, Port 0 and 2 continue to output whatever is in the P0 and P2 registers.

Termination from the Idle Mode

There are two ways to terminate Idle Mode :

- 1) Activation of any enable interrupt will cause the hardware to clear bit 0 of the PCON register, terminating the Idle Mode. After the execution of the interrupt service routine, program execution will resume from the next instruction that invoked Idle Mode.
- 2) The other way is with a hardware reset. Since the clock oscillator is still running, RST only needs to be held active for two machine cycles (24 oscillator periods) to complete the reset. This exit from Idle Mode writes 1s to all the ports, initializes all SFRs to their reset values, and restart program execution from location 0.

7.10.2 Power Down Mode

In the Power Down Mode ($PD = 0$), the CPU puts the whole chip to sleep by turning off the oscillator. In case if it is running from an external oscillator, it also gates off the path to the internal phase generators, so no internal clock is generated even if the external oscillator is still running. The on-chip RAM, however, saves its data, as long as V_{CC} is

maintained. In this mode the only I_{CC} that flows is leakage, which is normally in the micro-amp range.

The Power Down Mode is invoked by setting bit 1 in the PCON register, using a byte instruction such as

ORL PCON, #2

While the device is in Power Down Mode, ALE and \overline{PSEN} emit low (V_{OL}). The reason they are designed to output low is so that power can be removed from the reset of the circuit, if desired, while the 8051 is in its Power Down Mode.

The port pins continue to output whatever data was written to them. Note that port 2 emits its P2 register data even if execution was from external program memory.

Port 0 also outputs its P0 register data, but if execution was from external program memory, the P0 register data is FFH.

The Table 7.8 summarizes the status of pins in Idle and Power Down Modes.

| Pin | Internal execution | | External execution | |
|-------------------|--------------------|------------|--------------------|------------|
| | Idle | Power down | Idle | Power down |
| ALE | 1 | 0 | 1 | 0 |
| \overline{PSEN} | 1 | 0 | 1 | 0 |
| P0 | SFR Data | SFR Data | High-Z | High-Z |
| P1 | SFR Data | SFR Data | SFR Data | SFR Data |
| P2 | SFR Data | SFR Data | PCH | SFR Data |
| P3 | SFR Data | SFR Data | SFR Data | SFR Data |

Table 7.8 Status of pins in idle and power down modes

Note : 1. "SFR DATA" means the port pins output their internal register data.

2. "PCH" is the high byte of the program counter.

Termination from Power Down Mode :

The only exit from Power Down is a hardware reset. Since the oscillator was stopped, RST must be held active long enough for the oscillator to restart and stabilize. Then the reset function initializes all the Special Function Registers (ports, timers, etc.) to their reset values, and restarts the program from location 0. Therefore, timer reloads, interrupt enables, baud rates, port status, etc. need to be re-established. Reset does not affect the content of the on-chip data RAM. If V_{CC} was held during Power Down, the RAM data is still good.

Using Power Down Mode :

The software invoked Power Down feature offers a means of reducing the power consumption in some applications, V_{CC} to part of the system may be turned off during Power Down, so that even quiescent and standby currents are eliminated. But before switching off V_{CC} of any chip signal lines connected to the chip must be brought to a logic low, whether the chip in question is CMOS, NMOS, or TTL. CMOS pins have parasitic p-n junctions to V_{CC} , which will be forward biased if V_{CC} is reduced to zero while the pins are held at a logic high. NMOS pin 3 often have FETs that look like diodes to V_{CC} . TTL circuits may actually be damaged by an input high if $V_{CC} = 0$.

To ensure safe power down operation it is necessary that V_{CC} for secondary circuit must be shut off after the 8051 is in power down and V_{CC} for secondary circuit must be switched on before the 8051 is out of power down. The Fig. 7.23 shows a circuit that can be used to turn V_{CC} off to part of the system (secondary circuit) during power down.

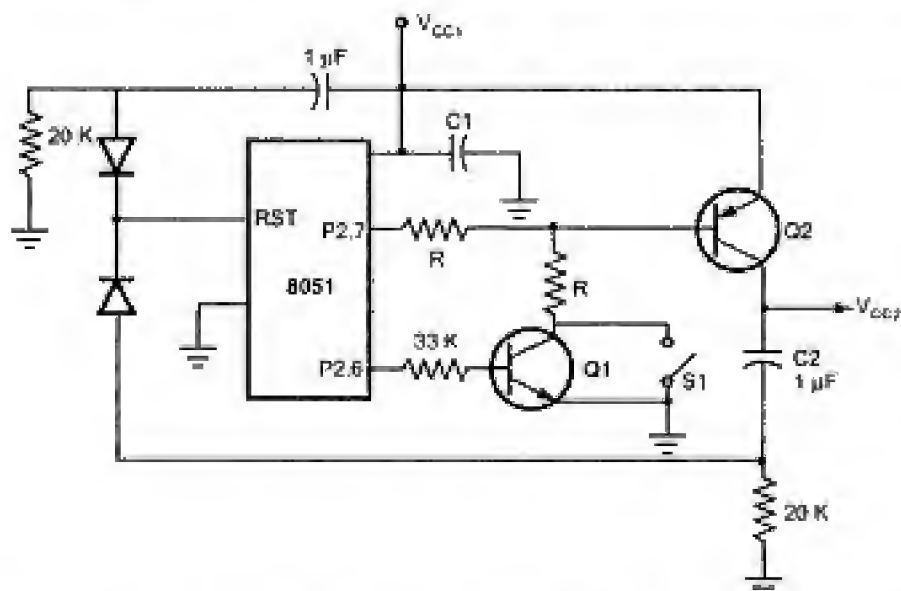


Fig. 7.23 Power switching circuit for power down mode

In Fig. 7.23 when V_{CC} is switched on to the 8051 capacitor $C1$ provides a power-on reset. The reset function writes 1s to all the port pins. The 1 at P2.6 turns Q1 on, enabling V_{CC} to the secondary circuit through transistor Q2. As the 8051 comes out of reset, Port 2 commences emitting the high byte of the program counter, which results in the P2.7 and P2.6 pins outputting 0s. The 0 at P2.7 ensures continuation of V_{CC} to the secondary circuit.

The system software must now write a 1 to P2.7 and a 0 to P2.6 in the Port 2 SFR, P2. These values will not appear at the Port 2 pins as long as the device is fetching instructions from external program memory. However, whenever the 8051 goes into Power Down, these values will appear at the port pins, and will shut off both transistors, disabling V_{CC} to the secondary circuit.

Closing the switch S1 re-energizes the secondary circuit, and at the same time sends a reset through C2 to the 8051 to wake it up. The diode D1 is to prevent C1 from hogging current from C2 during this secondary reset. D2 prevents C2 from discharging through the RST pin when V_{CC} to the secondary circuit goes to zero.

7.11 Multiprocessor Communication in MCS-51

In 8051 serial modes 2 and 3 have a special provision for multiprocessor communications. In these modes, 9 data bits are received. The 9th one goes into RB8. Then comes a stop bit. The port can be programmed such that when the stop bit is received, the serial port interrupt will be activated only if RB8 = 1. This feature is enabled by setting bit SM2 in SCON. A way to use this feature in multiprocessor systems is as follows.

When the master processor wants to transmit a block of data to one of several slaves, it first sends out an address byte which identifies the target slave. An address byte differs from a data byte in that the 9th bit is 1 in an address byte and 0 in a data byte. With SM2 = 1, no slave will be interrupted by a data byte. An address byte, however, will interrupt all slaves, so that each slave can examine the received byte and see if it is being addressed. The addressed slave will clear its SM2 bit and prepare to receive the data bytes that will be coming. The slaves that weren't being addressed leave their SM2s set and go on about their business, ignoring the coming data bytes.

Review Questions

1. *What is microcontroller?*
2. *What is the difference between microprocessor and microcontroller?*
3. *List the features of 8051 microcontroller.*
4. *Draw and explain the block diagram of 8051.*
5. *States various modes available for timer in 8051.*
6. *Explain serial communication modes supported by 8051.*
7. *Explain interrupt structure of 8051.*
8. *Explain the memory structure of 8051.*
9. *Explain the I/O port structure of 8051.*
10. *Explain the multiprocessor communication in 8051.*



Instruction Set and Programming of 8051

The 8051 architecture supports several distinct physical address spaces.

- a) On - chip program memory
- b) On - chip data memory
- c) External program memory
- d) External data memory
- e) On-chip special function registers

The most of the instructions can operate with variables from these physical address spaces.

In this chapter, we will see how to access variables from different physical and logical address spaces with the help of different addressing modes. Then we will study different instructions supported by 8051.

8.1 Addressing Modes

The way, using which the data sources or destination addresses are specified in the instruction mnemonic for moving the data, is called 'addressing mode'. This section explains addressing modes used in 8051 with examples.

1. Register Addressing

The 8051 can access eight "working registers" (R0-R7). Three bit code within the instruction selects one of the eight registers from the selected register bank. The programmer can select a register bank by modifying bits 4 and 3 in the PSW.



Example : Add the contents of register R3 and R4 from bank 2.

Step 1 : Select register bank.

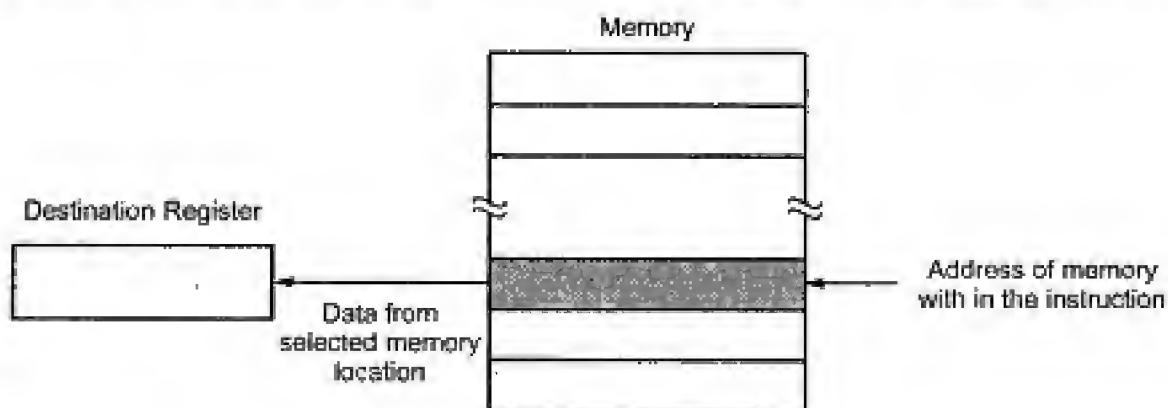
```
MOV PSW, #00001000B ; select register Bank 2
```

Step 2 : Add the contents of R3 and R4.

```
MOV A, R3
ADD A, R4
```

2. Direct Byte Addressing

Direct addressing can access any on chip variable or hardware register. i.e. on chip RAM and special function register. The most significant bit of the address decides whether it is a location within on chip RAM (MSB = 0) or in special function register (MSB = 1).

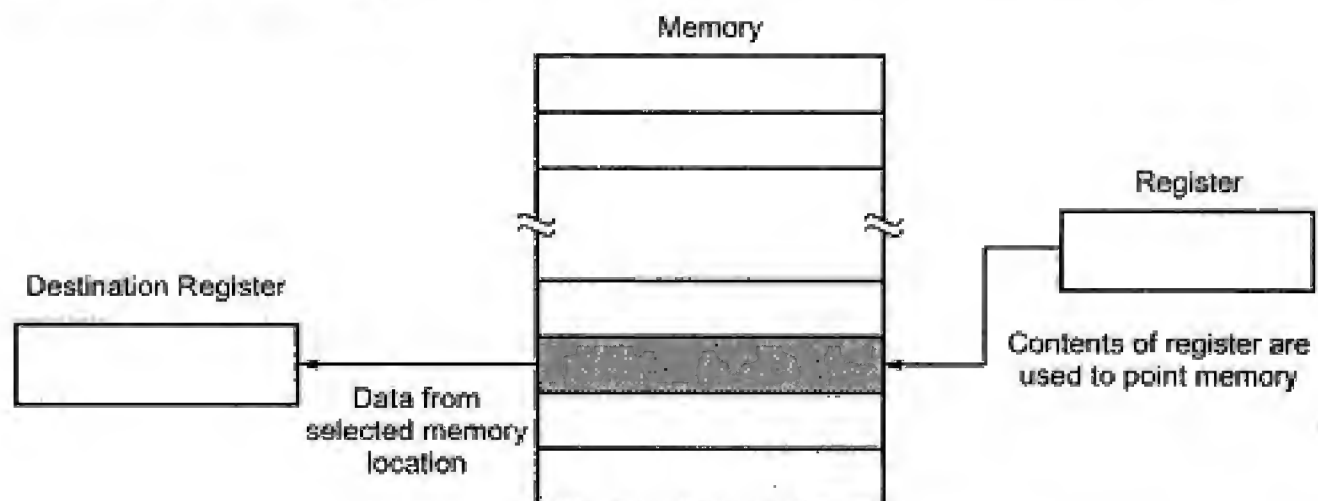


Example : Add the contents of locations 50H and 51H

```
MOV A, 50H           ; load byte from address 50H into A
ADD A, 51H           ; Add the contents of A and the contents
                    ; at memory location 51H.
```

3. Register Indirect Addressing

In this addressing mode R0 and R1 of each register bank can be used as an index or pointer register. R0 and R1 point to the contents in the RAM. The instruction with indirect addressing uses the '@' sign.



Example : *ADD the contents of memory location addressed by register 1 to the contents of RAM location pointed by register 0.*

```
MOV A, @R0 ; load the contents pointed by R0 in A.
ADD A, @R1 ; Add the contents of A and the contents pointed
            by R1
```

4. Immediate Addressing

In this addressing mode source operand is a constant rather than a variable. So the constant can be incorporated in the instruction. Sign “#” indicates it is a immediate addressing mode.



Example :

Add the constant 52 Decimal in Accumulator.

```
MOV A, #52
```

5. Register Specific

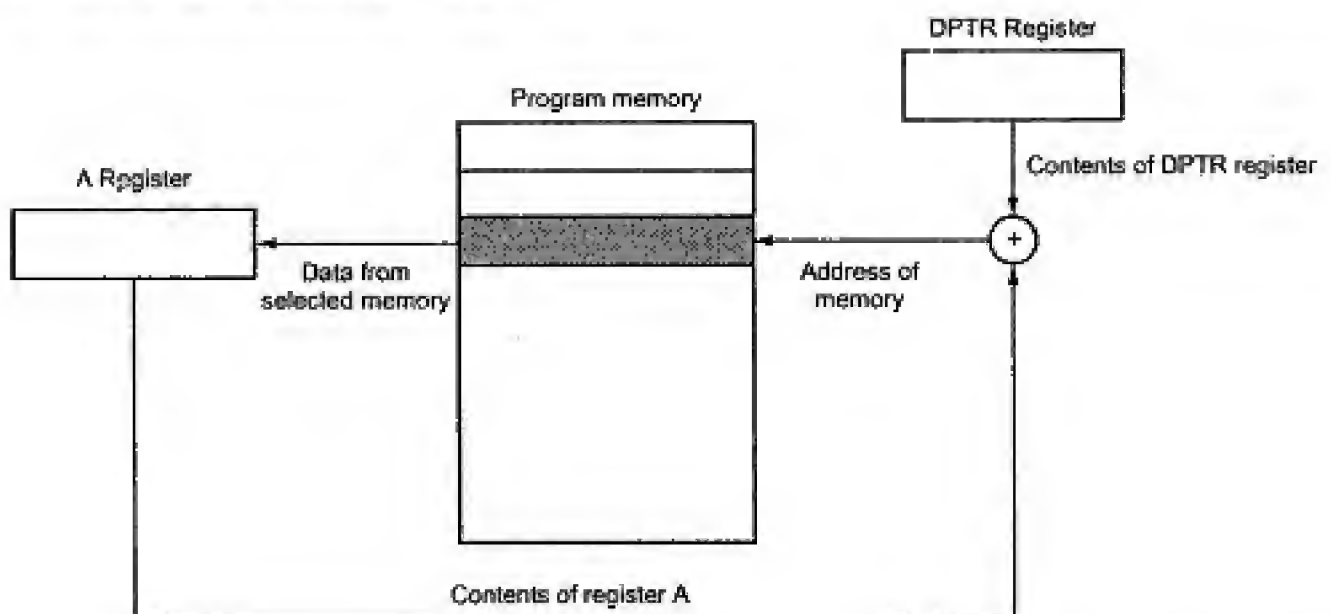
Inherent in the instruction, these refer to a specific register such as accumulator or DPTR.

Example :

```
SWAP A ; Swap nibbles within the Accumulator.
```

6. Index

Only program memory can be accessed in the index addressing. Either the DPTR or PC can be used as an index register.



Example : *Read data from the program memory.*

```
MOVC A, @A+DPTR ; This instruction adds the unsigned 8-bit
                  ; and accumulator contents into sixteen bit.
                  ; Data pointer, and uses the sum as an address
                  ; from which byte to be moved into accumulator.
```

8.2 Data Moving (Data Transfer) Instructions

An immediate, direct, register and indirect addressing modes are used in different MOVE instructions. Table 8.1 lists all types of data moving (data transfer) instructions.

| Mnemonic | Operation |
|-----------------------------|--|
| MOV <dest-byte>, <src-byte> | Copy the byte variable indicated by 'src-byte' into the 'dest-byte' location |
| MOV A, Rn | Copy the contents of register Rn of selected register bank to A. |
| MOV A, direct | Copy the contents of address specified with instruction to A. |
| MOV A, @ Ri | Copy the contents of the address in Rp to A. |
| MOV A, # data | Load data given in the instruction to A. |
| MOV Rn, A | Copy the contents of A to register Rn of selected register bank. |
| MOV Rn, direct | Copy the contents of address to register Rn of selected register bank. |
| MOV Rn, # data | Load data given in the instruction to register Rn of selected register bank. |
| MOV direct, A | Copy the contents A to the address specified within instruction. |
| MOV direct, Rn | Copy the contents of register Rn of selected bank register to the address specified within instruction. |
| MOV direct, direct | Copy the contents of the address specified within instruction to the address specified within instruction. |
| MOV direct, @ Ri | Copy the contents of the address given by register Ri of selected register bank to the address specified within instruction. |
| MOV direct, # data | Load data given within instruction to the address specified within instruction. |
| MOV @ Ri, A | Copy the contents of A to the address given by register Ri of selected register bank. |

| | |
|---------------------|--|
| MOV @ Ri, direct | Copy the contents of address specified within instruction to the address specified by register Ri of selected register bank. |
| MOV @ Ri, # data | Load the data specified within instruction to the address specified by register Ri of selected register bank. |
| MOV DPTR, # data 16 | Load data pointer with a 16-bit constant. |

Table 8.1 MOV instructions

The MOV instructions can be explained completely with example as given below.

| | | |
|---|----------------------|---------------------|
| MOV<dest-byte>, <src-byte> | Bytes : 1/2/3 | Cycles : 1/2 |
| Function : Move byte variable | | |

Description : The byte variable indicated by the second operand is copied into the location specified by the first operand. The source byte is not affected. No other register or flag is affected.

This is by far the most flexible operation. Fifteen combinations of source and destination addressing modes are allowed.

Example : Internal RAM location 30H holds 40H. The value of RAM location 40H is 10H. The data present at input port 1 is 11001010B (CAH).

```
MOV R0, #30H      ; R0 ← 30H
MOV A, @R0        ; A ← 40H
MOV R1, A         ; R1 ← 40H
MOV B, @R1        ; B ← 10H
MOV @R1, P1; RAM (40H) ← CAH
MOV P2, P1        ; P2 # CAH
```

leaves the value 30H in register 0, 40H in both the Accumulator and register 1, 10H in register B, and CAH (11001010B) both in RAM location 40H and output on port 2.

MOV A, Rn

Operation : (A) ← (Rn)

Example : MOV A, R0 ; This instruction copies the contents of the register R0 of selected register bank to the accumulator.

MOV A,direct

Example : **MOV A, 30H ;** This instruction copies the contents of memory location whose address is 30H to the accumulator.

Operation : $(A) \leftarrow (\text{direct})$

Example : **MOV A, 30H ;** This instruction copies the contents of memory location whose address is 30H to the accumulator.

MOV A, @Ri

Operation : $(A) \leftarrow ((Ri))$

Example : **MOVA, @R1 ;** This instruction copies the contents of memory location whose address is specified in the register R1 from selected register bank.

MOV A, #data

Operation : $(A) \leftarrow \# \text{ data}$

Example : **MOV A, # 30H ;** This instruction copies data given with in instruction (30H) into the accumulator.

MOV Rn, A

Operation : $(Rn) \leftarrow (A)$

Example : **MOV R2, A ;** This instruction copies the contents of accumulator in R2 register of selected register bank.

MOV Rn, direct

Operation : $(Rn) \leftarrow (\text{direct})$

Example : **MOV R1, 40H ;** This instruction copies the contents at memory address 40H into the R1 register of the selected register bank.

MOV Rn, # data

Operation : $(Rn) \leftarrow \# \text{ data}$

Example : **MOV R2, #20H ;** This instruction loads 20H in the registers R2 of selected register bank.

MOV direct, A

Operation : $(\text{direct}) \leftarrow (A)$

Example : **MOV 20H, A ;** This instruction copies the contents of accumulator to the direct memory address specified in the instruction (20H).

MOV direct, Rn

Operation : $(\text{direct}) \leftarrow (Rn)$

Example : **MOV 30H, R2 ;** This instruction copies the contents of register R2 of selected register bank to the direct memory address specified in the instruction (30H).

MOV direct, direct

Operation : $(\text{direct}) \leftarrow (\text{direct})$

Example : **MOV 20H, 40H ;** This instruction copies the contents of memory location whose address is 40H to the memory location whose address is 20H.

MOV direct, @Ri

Operation : $(\text{direct}) \leftarrow (Ri)$

Example : **MOV 20H, @R3 ;** This instruction copies the contents of memory location whose address is given by register R3 of selected register bank into the memory location whose address is 20H.

MOV direct, #data

Operation : $(\text{direct}) \leftarrow \#data$

Example : **MOV 30H, #12H ;** This instruction copies data given with in instruction (12H) into the memory location whose address is 30H.

MOV @Ri, A

Operation : $((Ri)) \leftarrow (A)$

Example : **MOV @R1, A ;** This instruction copies the contents of accumulator to the memory location whose address is given by register R1 of selected register bank.

MOV @Ri, direct

Operation : $(Ri) \leftarrow (\text{direct})$

Example : **MOV @ R2, 30H ;** This instruction copies the contents of memory location whose address is given within the instruction (30H) into

the memory location whose address is specified by register R2 of selected register bank.

MOV @Ri, #data

Operation : $((Ri)) \leftarrow \#data$

Example : **MOV @R2, #30H** ; This instruction loads 30H into the memory location whose address is specified by register R2 of selected register bank.

MOV A, ACC is not a valid instruction.

MOV DPTR, #data16

Bytes : 3

Cycles : 2

Function : Load Data Pointer with a 16-bit constant

Description : The Data pointer is loaded with the 16-bit constant indicated. The 16-bit constant is loaded into the second and third bytes of the instruction. The second byte (DPH) is the high-order byte, while the third byte (DPL) holds the low-order byte. No flags are affected.

This is the only instruction which moves 16 bits of data at once.

Example : The instruction,

MOV DPTR, #1234H

will load the value 1234H into the Data Pointer: DPH will hold 12H and DPL will hold 34H.

Operation : **MOV**

$(DPTR) \leftarrow \#data\ 15-0$

$DPH \leftarrow \#data15-8\ DPL \leftarrow \#data7-0$

8.2.1 Instructions to Access External Data Memory

The Table 8.2 explains the instruction to access external data memory.

| Mnemonic | Operation |
|---------------|---|
| MOVX A, @Ri | Copy the contents of the external address in Ri to A. |
| MOVX A, @DPTR | Copy the contents of the external address in DPTR to A. |

| | |
|---------------|---|
| MOVX @ Ri, A | Copy data from A to the external address in Ri. |
| MOVX @DPTR, A | Copy data from A to the external address in DPTR. |

Table 8.2

Some examples of the instructions listed in Table 8.2 are given below.

Example 1 : MOVX A, @ R0

This instruction copies data from the 8-bit address in R0 to A.

Example 2 : MOVX A, @ DPTR

This instruction copies data from the 16-bit address in DPTR to A.

Example 3 : MOVX @ R1, A

This instruction copies data from A to the 8-bit address in R1.

Example 4 : MOVX @ DPTR, A

This instruction copies data from A to the 16-bit address in DPTR.

The MOVX instructions can be explained completely with example as given below.

MOVX <dest-byte>,<src-byte> Function : Move External Bytes : 1 Cycles : 2

Description : The MOVX instructions transfer data between the Accumulator and a byte of external data memory, hence the "X" appended to MOV. There are two types of instructions, differing in whether they provide an eight-bit or sixteen-bit indirect address to the external data RAM. In the first type, the contents of R0 or R1 in the current register bank provide an eight-bit address multiplexed with data on P0. Eight bits are sufficient for external I/O expansion decoding or for a relatively small RAM array. For somewhat larger arrays, any output port pins can be used to output higher-order address bits. These pins would be controlled by an output instruction preceding the MOVX. In the second type of MOVX instruction, the Data Pointer generates a sixteen-bit address. P2 outputs the high-order eight address bits (the contents of DPH) while P0 multiplexes the low order eight bits (DPL) with data. The P2 output buffers are emitting the contents of DPH. This form is faster and more efficient when accessing very large data arrays (up to 64 Kbytes), since no additional instructions are needed to set up the output ports.

It is possible in some situations to mix the two MOVX types. A large RAM array with its high-order address lines driven by P2 can

be addressed via the Data Pointer, or with code to output high-order address bits to P2 followed by a MOVX instruction using R0 or R1.

Example :

An external 256 byte RAM using multiplexed address/data lines (e.g., an Intel 8155 RAM/ I/O/ Timer) is connected to the 8051 Port 0. Port 3 provides control lines for the external RAM. Ports 1 and 2 are used for normal I/O. Register 0 and 1 contain 12H and 34H. Location 34H of the external RAM holds the value 56H. The instruction sequence,

```
MOVX    A, @ R1
```

```
MOVX    @ R0,A
```

copies the value 56H into both the Accumulator and external RAM location 12H.

MOVX A, @Ri

This instruction copies the contents of external memory whose address is given by register into the accumulator.

Operation :

$(A) \leftarrow ((Ri))$

MOVX A, @DPTR :

This instruction copies the contents of external memory whose address is given by DPTR register into the accumulator.

Operation :

$(A) \leftarrow ((DPTR))$

MOVX @Ri, A :

This instruction copies the contents of accumulator into the external memory whose address is given by the register.

Operation :

$((Ri)) \leftarrow (A)$

MOVX @DPTR, A :

This instruction copies the contents of accumulator into the external memory whose address is given by the DPTR register.

Operation :

$((DPTR)) \leftarrow (A)$

Important Points to Remember In Accessing External Data Memory

- All external data moves with external RAM involve the A register.
- While accessing external RAM, R_p can address 256 bytes and DPTR can address 64 kbytes.
- MOVX instruction is used to access external RAM or I/O addresses.

There are two sets of RAM addresses between 00H and FFH, one for internal 8051 RAM and another for RAM external to 8051.

8.2.2 Instructions to Access External ROM / Program Memory

The Table 8.3 explains the instructions to access external ROM/program memory.

| Mnemonic | Operation |
|--------------------|--|
| MOVC A, @ A + DPTR | Copy the contents of the external ROM address formed by adding A and the DPTR, to A. |
| MOVC A, @ A + PC | Copy the contents of the external ROM address formed by adding A and the PC, to A. |

Table 8.3

Example 1 : Let the contents of DPTR are 1200 and the contents of A are 61H.
MOVC A, @ A + DPTR

This instruction copies the code byte found at the external ROM address formed by adding A and the DPTR, i.e. at an address (1200 H + 61 H) 1261 H to A.

Example 2 : Let the contents of PC are 4000 H and contents of A are 50H.

MOVC A, @ A + PC

This instruction copies the code byte found at the external ROM address formed by adding A and the PC, i.e. at an address (4000 H + 50 H) 4050 H to A.

The MOVC instructions can be explained completely with example as given below.

MOVC A,@A+<base-reg> Function : Move Code byte Bytes : 1 Cycles : 2

Description : The MOVC instructions load the Accumulator with a code byte, or constant from program memory. The address of the byte fetched is the sum of the original unsigned eight-bit Accumulator contents and the contents of a sixteen-bit base register, which may be either the Data Pointer or the PC. In the latter case, the PC is incremented to the address of the following instruction before being added with the Accumulator; otherwise the base register is not altered. Sixteen-bit addition is performed so a carry-out from the low-order eight bits may propagate through higher-order bits. No flags are affected.

Example : A value between 0 and 3 is in the Accumulator. The following instructions will translate the value in the Accumulator to one of four values defined by the DB (define byte) directive.

```
REL_PC :      INC  A
              MOVC A,@A+PC
              RET
              DB   66H
              DB   77H
              DB   88H
              DB   99H
```

If the subroutine is called with the Accumulator equal to 01H, will return with 77H in the Accumulator. The INC A before the move instruction is needed to "get around" the RET instruction above the table. If several bytes of code separated the MOVC from the table, the corresponding number would be added to the Accumulator instead.

MOVC A,@A+DPTR : This instruction loads the accumulator from the contents of program memory whose address is given by the sum of the contents of accumulator and contents of DPTR register.

Operation : $(A) \leftarrow ((A) + (DPTR))$

MOVC A,@A + PC : This instruction loads the accumulator from the contents of program memory whose address is given by the sum of the contents of accumulator and the contents of program counter. The current contents of program counter are incremented by 1 before summation

Operation : $(PC) \leftarrow (PC) + 1$
 $(A) \leftarrow ((A) + (PC))$

Important Points to Remember in Accessing External Read Only Memory

- When PC is used to access external ROM, it is incremented by 1 (to point to the next instruction) before it is added to A to form the physical address of external ROM.
- All external data moves with external ROM involve the A register.
- MOVC is used with internal or external ROM and can address 4 K of internal code or 64 K of external code.
- The DPTR and the PC are not changed.

8.2.3 PUSH and POP Instructions

The stack refers to an area of internal RAM that is used in conjunction with certain opcodes to store and retrieve data quickly. The stack pointer, (SP) is an 8-bit register used by the 8051 to hold stack address that is called **top of stack**. The SP is set to 07 H when the 8051 is reset. In PUSH and POP operations, it is implied that the SP holds the indirect address.

8.2.3.1 PUSH

The PUSH instruction increments the stack pointer by one and copies data from the source address (specified in the PUSH instruction) to the internal RAM location addressed by the stack pointer. Before copy operation, since SP is incremented, the data is stored from low addresses to high addresses in the internal RAM. As data is pushed, the stack grows up in internal RAM. The top of the internal RAM is at 7FH. So if we go on PUSHing the data onto the stack after 7FH, the data is lost. Fig. 8.1 shows the PUSH operation.

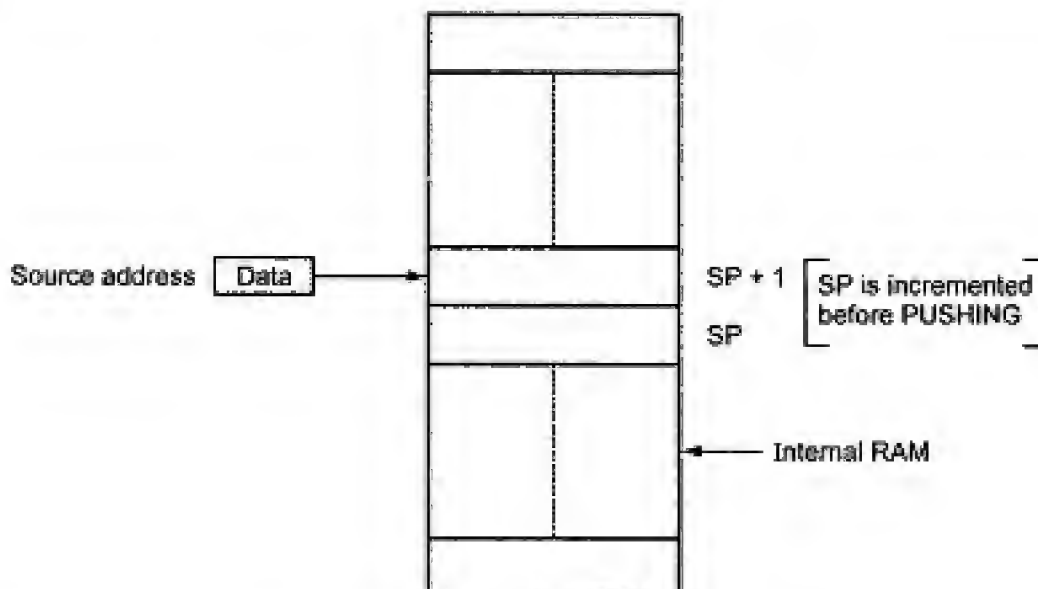


Fig. 8.1 PUSH operation

Instruction : PUSH source address

Example : PUSH B

This instruction increments the stack pointer by one and stores the contents of register B to the internal RAM location addressed by the stack pointer (SP).

The PUSH instruction can be explained completely with example as given below.

| PUSH direct | Function : Push onto stack | Bytes : 2 | Cycles : 2 |
|--------------------|-----------------------------------|------------------|-------------------|
|--------------------|-----------------------------------|------------------|-------------------|

Description : The Stack Pointer is incremented by one. The contents of the indicated variable is then copied into the internal RAM location addressed by the Stack Pointer. Otherwise no flags are affected.

Example : On entering an interrupt routine the Stack Pointer contains 09H. The Data Pointer holds the value 0123H. The instruction sequence,
 PUSH DPL
 PUSH DPH
 will leave the Stack Pointer set to 0BH and store 23H and 01H in internal RAM locations 0AH and 0BH, respectively.

Operation : $(SP) \leftarrow (SP) + 1$
 $((SP)) \leftarrow (\text{direct})$

8.2.3.2 POP

The POP instruction copies the contents of the internal RAM location addressed by the stack pointer to the destination address (specified in the POP instruction). Then the stack pointer is decremented by one. This ensures that data placed on the stack is retrieved in the same order as it was stored. Fig. 8.2 shows the POP operation.

Instruction : POP destination address

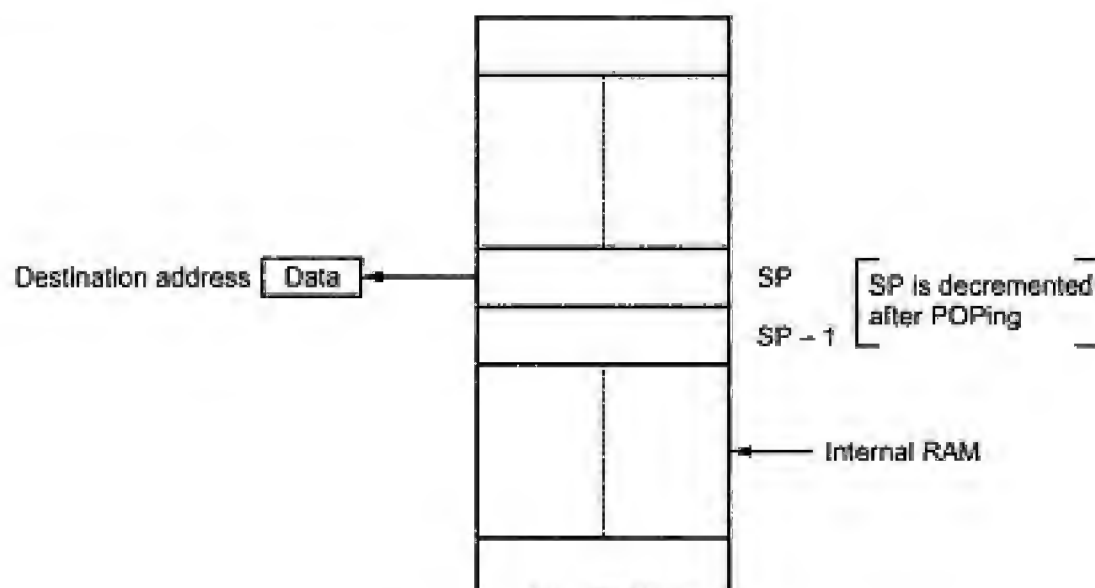


Fig. 8.2 POP operation

Example : POP ACC

This instruction copies the contents of the internal RAM location addressed by the stack pointer to the accumulator. Then the stack pointer is decremented by one.

The POP instruction can be explained completely with example as given below.

| POP direct | Function : Pop from stack | Bytes : 2 | Cycles : 2 |
|------------|---------------------------|-----------|------------|
|------------|---------------------------|-----------|------------|

Description : The contents of the internal RAM location addressed by the Stack Pointer is read, and the Stack Pointer is decremented by one. The value read is then transferred to the directly addressed byte indicated. No flags are affected.

Example : The Stack Pointer originally contains the value 32H, and internal RAM locations 30H through 32H contain the values 20H, 23H, and 01H, respectively. The instruction sequence,

POP DPH

POP DPL

will leave the stack Pointer equal to the value 30H and the Data Pointer set to 012H. At this point the instruction,

POP SP

will leave the Stack Pointer set to 20H. Note that in this special case the Stack Pointer was decremented to 2FH before being loading with the value popped (20H).

Operation : (direct) \leftarrow ((SP))

(SP) \leftarrow (SP) - 1

Important Points to Remember during PUSH and POP

- When the SP contents become FFH, for the next PUSH, the SP rolls over to 00H.
- The top of the internal RAM, i.e. it's end address is 7FH. So next PUSHes after 7FH result in errors.
- Generally the SP is set at address above the register banks.
- The PUSH and POP operations may be applied to the stack pointer (SP).
- When PUSH and POP operations are used for the registers from the register banks (bank 0 - bank 3), specify direct addresses within the instructions. Do not use register name from register bank since the register name does not specify the bank in use.

8.2.4 Data Exchange Instructions

When 8051 executes MOV, PUSH or POP instruction, the 'copy operation' takes place. The data from the source address is copied to the destination address. The data at the source address remains unchanged. The Exchange instructions move data from source address to destination address and vice versa.

Table 8.4 lists all types of exchange instructions in 8051.

| Mnemonic | Operation |
|---------------|--|
| XCH A, <byte> | Exchange Accumulator with byte variable. |
| XCH A, Rn | Exchange data bytes between register Rn and A. |
| XCH A, direct | Exchange data bytes between address directly given within instruction and A. |
| XCH A, @ Ri | Exchange data bytes between A and address in Ri. |
| XCHD A, @ Ri | Exchange lower nibble between A and address in Ri. |

Table 8.4 Exchange Instructions

All the instructions given in the Table 8.4 are explained completely with example as below.

XCH A,<byte> Bytes : 1/2 Cycles : 1 Function : Exchange Accumulator with byte variable

Description : XCH loads the Accumulator with the contents of the indicated variable, at the same time writing the original Accumulator contents to the indicated variable. The source/destination operand can use register, direct, or register-indirect addressing.

Example : R0 contains the address 20H. The Accumulator holds the value 3FH (00111111B). Internal RAM location 20H holds the value 75H (01110101B). The instruction,

XCH A, @R0

will leave RAM location 20H holding the values 3FH (00111111B) and 75H (01110101B) in the accumulator.

XCH A, Rn

Operation : $(A) \xleftrightarrow{\leftarrow} (R_n)$

Example : XCH A, R0 ; This instruction exchanges contents of accumulator with the contents of register R0 of selected register bank.

XCH A, direct

Operation : $(A) \xrightarrow{\quad} (direct)$
 $\quad \quad \quad \leftarrow$

Example : **XCH A, 20H ;** This instruction exchanges contents of accumulator with the contents of memory whose address is given within the instruction (20H)

XCH A, @Ri

Operation : $(A) \xrightarrow{\quad} (R_i)$
 $\quad \quad \quad \leftarrow$

Example : **MOV A, #R2 ;** This instruction exchanges the contents of accumulator with the contents of memory location whose address is given by the contents of register R2 of selected register bank.

| XCHD A, @Ri | Function : Exchange Digit | Bytes : 1 | Cycles : 1 |
|----------------------|---|------------------|-------------------|
| Description : | XCHD exchanges the low-order nibble of the Accumulator (bits 3-0), generally representing a hexadecimal or BCD digit, with that of the internal RAM location indirectly addressed by the specified register. The high-order nibbles (bits7-4) of each register are not affected. No flags are affected. | | |
| Example : | <p>R0 contains the address 20H. The Accumulator holds the value 36H (00110110B). Internal RAM location 20H holds the value 75H (01110101B) The instruction,</p> <p>XCHD A, @R0</p> <p>will leave RAM location 20H holding the value 76H (01110110B) and 35H (00110101B) in the Accumulator.</p> | | |
| Operation : | $(A_{3-0}) \xrightarrow{\quad} (R_{i3-0})$ $\quad \quad \quad \leftarrow$ | | |

Important Points to Remember in Exchange Instructions

- All exchanges involve the A register.
- All exchanges take place internally within 8051.
- When XCHD A, @Ri instruction is executed, the upper nibble of A and the upper nibble of the address in Ri do not change.
- Immediate addressing mode can not be used in the exchange instructions.

8.3 Logical Instructions

In this section we are going to study byte and bit level logical operations and rotate and swap operations.

Byte and bit level logical instructions operate on data using Boolean operators. They can be used to manipulate data in all 8051 RAM areas (both data and SFRs). We know that, many of the SFRs, and a unique internal RAM is bit addressable. Such bit addressable locations can be efficiently manipulated by bit level instructions. Bit level instructions yield compact program code that enhances program execution speed.

The Table 8.5 gives the mnemonic for various logical operations supported by 8051.

| Boolean Operator | Mnemonic in 8051 |
|------------------|------------------|
| AND | ANL |
| OR | ORL |
| XOR | XRL |
| NOT | CPL |

Table 8.5

The 8051 also supports rotate instructions. These instructions operate only on a byte, or a byte and the carry flag to permit limited 8 and 9-bit shift-register operations. The Table 8.6 shows the mnemonic for various rotate operations supported by 8051.

| Rotate operation | Mnemonic |
|---|----------|
| Rotate byte left | RL |
| Rotate byte left through carry | RLC |
| Rotate byte right | RR |
| Rotate byte right through carry | RRC |
| Exchange the low and high nibbles in a byte | SWAP |

Table 8.6

8.3.1 Byte Level Logical Operations

The instructions ANL, ORL, and XRL perform the logical functions AND, OR, and/or Exclusive-OR on the two byte variables indicated, leaving the results in the first. No flags are affected.

The byte-level logical operations use all four addressing modes for the source of a data byte. Here, directly-addressed bytes may be used as the destination with either the accumulator or a constant as the source. These instructions are useful for clearing (ANL), setting (ORL), or complementing (XRL) one or more bits in a RAM, output ports, or control registers.

This is illustrated in Fig. 8.3.

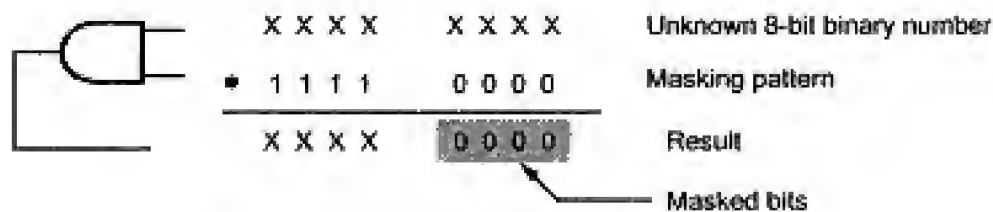


Fig. 8.3 Masking using AND operation

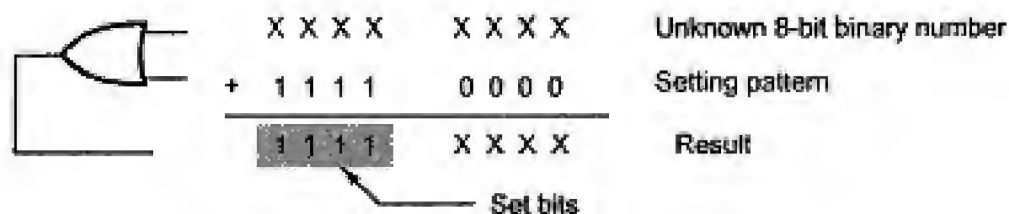


Fig. 8.4 Setting bit/s using OR operation

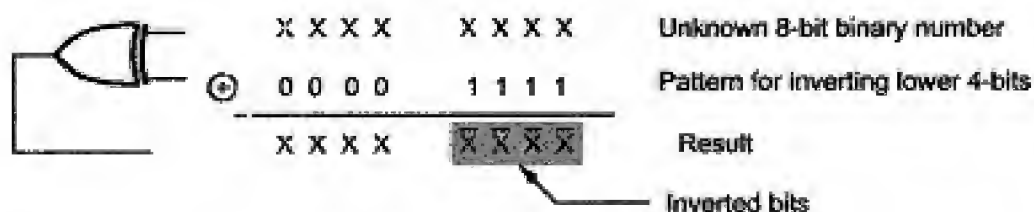


Fig. 8.5 Inversion of part of a number using XOR operation

The Table 8.7 gives the list of byte level logical operations.

| Mnemonic | | Destination | Byte | Cyc |
|----------|--------------|---|------|-----|
| ANL | A,Rn | AND register to Accumulator | 1 | 1 |
| ANL | A,direct | AND direct byte to Accumulator | 2 | 1 |
| ANL | A,@Ri | AND indirect RAM to Accumulator | 1 | 1 |
| ANL | A,#data | AND immediate data to Accumulator | 2 | 1 |
| ANL | direct,A | AND Accumulator to direct byte | 2 | 1 |
| ANL | direct,#data | AND immediate data to direct byte | 3 | 2 |
| ORL | A,Rn | OR register to Accumulator | 1 | 1 |
| ORL | A,direct | OR direct byte to Accumulator | 2 | 1 |
| ORL | A,@Ri | OR indirect RAM to Accumulator | 1 | 1 |
| ORL | A,#data | OR immediate data to Accumulator | 2 | 1 |
| ORL | direct,A | OR Accumulator to direct byte | 2 | 1 |
| ORL | direct,#data | OR immediate data to direct byte | 3 | 2 |
| XRL | A,Rn | Exclusive-OR register to Accumulator | 1 | 1 |
| XRL | A,direct | Exclusive-OR direct byte to Accumulator | 2 | 1 |
| XRL | A,@Ri | Exclusive-OR indirect RAM to A | 1 | 1 |
| XRL | A,#data | Exclusive-OR immediate data to A | 2 | 1 |
| XRL | direct,A | Exclusive-OR Accumulator to direct byte | 2 | 1 |
| XRL | direct,#data | Exclusive-OR immediate data to direct | 3 | 2 |
| CLR | A | Clear Accumulator | 1 | 1 |
| CPL | A | Complement Accumulator | 1 | 1 |

Table 8.7 Logical operations

Byte Level Logical Instructions with Examples

ANL<dest-byte>, <src-byte> Function : Logical-AND Bytes : 1/2/3 Cycles : 1

Description : ANL performs the bitwise logical-AND operation between the variables indicated and stores the result in the destination variable. No flags are affected.

The two operands allow six addressing mode combinations. When the destination is the Accumulator, the source can use register, direct, register-indirect, or immediate addressing; when the destination is a direct address, the source can be the Accumulator or immediate data.

Note : When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, not the input pins.

Example : If the Accumulator holds 89H (100001001B) and register 0 holds 95H (10010101B) in the Accumulator.

ANL A, R0

will leave 41H (01000001B) in the Accumulator.

When the destination is a directly addressed byte, this instruction will clear combinations of bits in any RAM location or hardware register.

The instruction,

ANL P1, #01110011B

will clear bits 7, 3, and 2 of output port 1.

ANL A, Rn

Operation : $(A) \leftarrow (A) \wedge (R_n)$

Example : ANL A, R2 ; Logically ANDs A and R2 and store result in A.

ANL A, direct

Operation : $(A) \leftarrow (A) \wedge (\text{direct})$

Example : ANL A, 20H ; Logically ANDs contents of A and memory location whose address is 20H and stores result in A.

ANL A, @R1

Operation : $(A) \leftarrow (A) \wedge ((R_1))$

Example : ANL A, @R2 ; Logically ANDs contents of A and memory location whose address is given by R2 and stores result in A.

ANLC A, #data

Operation : $(A) \leftarrow (A) \wedge \#data$

Example : ANL A, # 50H ; logically ANDs contents of A with 50H and stores result in A.

ANL direct, A

Operation : $(direct) \leftarrow (direct) \wedge (A)$

Example : ANL 20H, A ; Logically ANDs contents of A with the contents of memory location 20H and stores result at memory location 20H.

ANL direct, #data

Operation : $(direct) \leftarrow (direct) \wedge \#data$

Example : ANL 20H, #20H ; Logically ANDs the contents of memory location 20H with data 20H and stores result in memory location 20H.

ORL <dest-byte> <src-byte>

Bytes : 1/2/3

Cycles : 1/2

Function : Logical-OR for byte variables

Description : ORL performs the bitwise logical-OR operation between the indicated variables, storing the results in the destination byte. No flags are affected.

The two operands allow six addressing mode combinations. When the destination is the Accumulator, the source can use register, direct, register-indirect, or immediate addressing; when the destination is a direct address, the source can be the Accumulator or immediate data.

Note : when this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, not the input pins.

Example : If the Accumulator holds C3H (11000011B) and R0 holds 55H (01010101B) then the instruction,

ORL A, R0

will leave the Accumulator holding the value D7H (11010111B).

When the destination is a directly addressed byte, the instruction can set combinations of bits in any RAM location or hardware register. The pattern of bits to be set is determined by a mask byte, which may be either a constant data value in the instruction or a variable computed in the Accumulator at run-time. The instruction,

ORL P1, #00110010B

will set bits 5,4, and 1 of output Port 1.

ORL A, Rn

Operation : $(A) \leftarrow (A) \vee (Rn)$

Example : **ORL A, R2 ;** Logically ORs the contents of A and R2 and stores result in A.

ORL A, direct

Operation : $(A) \leftarrow (A) \vee (\text{direct})$

Example : **ORL A, 20H ;** logically ORs the contents of A and memory location 20H and stores result in A.

ORL A, @Ri

Operation : $(A) \leftarrow (A) \vee ((Ri))$

Example : **ORL A, @R2 ;** Logically ORS the contents of A and memory location whose address is given by register R2 and stores result in A.

ORL A, #data

Operation : $(A) \leftarrow (A) \vee \#data$

Example : **ORL A, #32H ;** Logically ORs the contents of A with 32H and stores result in A

ORL direct, A

Operation : $(\text{direct}) \leftarrow (\text{direct}) \vee (A)$

ORL direct, #data

Operation : $(\text{direct}) \leftarrow (\text{direct}) \vee \#data$

Example : **ORL 20H, #30H ;** Logically ORs the contents of memory location 20H and data 30H and stores result at memory location 20H.

XRL <dest-byte>,<src-byte>**Bytes : 1/2 Cycles : 1****Function : Logical Exclusive-OR for byte variables**

Description : XRL performs the bitwise logical Exclusive-OR operation between the indicated variables, storing the results in the destination. No flags are affected.

The two operands allow six addressing mode combinations. When the destination is the Accumulator, the source can use register, direct, register-indirect, or immediate addressing; when the destination is a direct address, the source can be the Accumulator or immediate data.

(Note : When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, not the input pins.)

Example : If the Accumulator holds C3H (11000011B) and register 0 holds AAH (10101010B) then the instruction.

XRL A, R0

will leave the Accumulator holding the value 69H (01101001B).

When the destination is a directly addressed byte, this instruction can complement combinations of bits in any RAM location or hardware register. The pattern of bits to be complemented is then determined by a mask byte, either a constant contained in the instruction or a variable computed in the Accumulator at run-time. The instruction,

XRL P1, # 00110001B

will complement bits 5, 4 and 0 of output Port 1.

XRL A, Rn

Operation : $(A) \leftarrow (A) \vee (Rn)$

Example : XRL A, R2 ; logically XOR the contents of A and R2 and stores result in A.

XRL A, direct

Operation : $(A) \leftarrow (A) \vee (\text{direct})$

Example : XRL A, 20H ; Logically XORs the contents of A with memory location 20H and stores result in A.

XRL A, @Ri**Operation :** $(A) \leftarrow (A) \vee ((Ri))$ **Example :** XRL A,@R2 ; Logically XORs the contents of A and the memory location whose address is given by R2 and stores result in A.**XRL A, #data****Operation :** $(A) \leftarrow (A) \vee \# \text{ data}$ **Example :** XRL A, #40H ; Logically XORs the contents of A with data 40H and stores result in A.**XRL direct, A****Operation :** $(\text{direct}) \leftarrow (\text{direct}) \vee (A)$ **Example :** XRL 20H, A ; Logically XORs the contents at 20H and the A and stores the result at 30H.**XRL direct, #data****Operation :** $(\text{direct}) \leftarrow (\text{direct}) \vee \# \text{ data}$ **Example :** XRL 30H, #40H ; Logically XORs the contents at 30H and data 40H and stores the result at 30H.

| CLR A | Function : Clear Accumulator | Bytes : 1 | Cycles : 1 |
|-------|------------------------------|-----------|------------|
|-------|------------------------------|-----------|------------|

Description : The Accumulator is cleared (all bits set on zero). No flags are affected.

Example : The Accumulator contains 95H (10010101B). The instruction,
CLR A
will leave the Accumulator set to 00H (00000000B).

Operation : $(A) \leftarrow 0$

| CPL A | Function : Complement Accumulator | Bytes : 1 | Cycles : 1 |
|-------|-----------------------------------|-----------|------------|
|-------|-----------------------------------|-----------|------------|

Description : Each bit of the Accumulator is logically complemented (one's complement). Bits which previously contained a one are changed to a zero and vice-versa. No flags are affected.

Example : The Accumulator contains 55H (01010101B). The instruction,

CPL A

will leave the Accumulator set to AAH (10101010B).

Operation : $(A) \leftarrow \overline{(A)}$

8.3.2 Bit Level Logical Operations

Bit level manipulations are very convenient when it is necessary to set or reset a particular bit in the internal RAM or SFRs. The internal RAM of 8051 from address 20H through 2FH is both byte addressable and bit addressable. However, byte and bit addresses are different. The Table 8.8 shows the correspondence between byte and bit addresses.

| Byte Address in Hex | Bit Address in Hex |
|---------------------|--------------------|
| 20 | 00-07 |
| 21 | 08-0F |
| 22 | 10-17 |
| 23 | 18-1F |
| 24 | 20-27 |
| 25 | 28-2F |
| 26 | 30-37 |
| 27 | 38-3F |
| 28 | 40-47 |
| 29 | 48-4F |
| 2A | 50-57 |
| 2B | 58-5F |
| 2C | 60-67 |
| 2D | 68-6F |
| 2E | 70-77 |
| 2F | 78-7F |

Table 8.8 Bit and byte addresses of internal RAM

As shown in the Table 8.8, addresses of bit 0 and bit 7 of internal RAM byte address 20H are 00H and 07H respectively. From Table 8.8 we can easily interpolate addresses of bit 1 and bit 6 of internal RAM byte address 26H as 31H and 36H, respectively.

Like internal RAM, some SFRs are bit addressable. The Table 8.9 shows the bit addressable SFR and the corresponding bit addresses.

| SFR | Direct Address in Hex | Bit Address in Hex |
|------|-----------------------|--------------------|
| A | E0 | E0-E7 |
| B | F0 | F0-F7 |
| IE | A8 | A8-AF |
| IP | B8 | B8-BF |
| P0 | 80 | 80-87 |
| P1 | 90 | 90-97 |
| P2 | A0 | A0-A7 |
| P3 | B0 | B0-B7 |
| PSW | D0 | D0-D7 |
| TCON | 88 | 88-8F |
| SCON | 98 | 98-9F |

Table 8.9 Bit and byte addresses of SFRs

The Table 8.10 gives the list of bit level operations.

| Mnemonic | Operands | Description | Byte | Cyc |
|----------|----------|---------------------------------------|------|-----|
| CLR | C | Clear Carry flag | 1 | 1 |
| CLR | bit | Clear direct bit | 2 | 1 |
| SETB | C | Set Carry flag | 1 | 1 |
| SETB | bit | Set direct bit | 2 | 1 |
| CPL | C | Complement Carry flag | 1 | 1 |
| CPL | bit | Complement direct bit | 2 | 1 |
| ANL | C,bit | AND direct bit to Carry flag | 2 | 2 |
| ANL | C,/bit | AND complement of direct bit to Carry | 2 | 2 |
| ORL | C,bit | OR direct bit to Carry flag | 2 | 2 |
| ORL | C,/bit | OR complement of direct bit to Carry | 2 | 2 |
| MOV | C,bit | Move direct bit to Carry flag | 2 | 1 |
| MOV | bit,C | Move Carry flag to direct bit | 2 | 2 |

Table 8.10 Boolean variable manipulation

Bit Level Instructions with Examples

| CLR bit | Function : Clear bit | Bytes : 1 | Cycles : 1 |
|---------------|---|-----------|------------|
| Description : | The indicated bit is cleared (reset to zero). No other flags are affected. CLR can operate on the carry flag or any directly addressable bit. | | |
| Example : | Port 1 has previously been written with FFH (11111111B). The instruction, CLR P1.2 will leave the port set to FBH (11111011B) | | |
| CLR C | | | |
| Operation : | (C) ← 0 | | |
| CLR bit | | | |
| Operation : | CLR (bit) ← 0 | | |

| SETB <bit> | Function : Set Bit | Bytes : 1/2 | Cycles : 1 |
|----------------------|--|-------------|------------|
| Description : | SETB sets the indicated bit to one. SETB can operate on the carry flag or any directly addressable bit. No other flags are affected. | | |
| Example : | The carry is cleared. Output Port 1 has been written with the value 34 H (00110100B). The instruction, SETB C SETB P1.0 will leave the carry flag set to 1 and change the data output on Port 1 to 35H (00110101B). | | |
| SETB C | | | |
| Operation : | (C) ← 1 | | |
| SETB bit | | | |
| Operation : | (bit) ← 1 | | |

| CPL bit | Function : Complement bit | Bytes : 1/2 | Cycles : 1 |
|------------------------------|---|------------------|-------------------|
| Description : | The bit variable specified is complemented. A bit which had been a one is changed to zero and vice-versa. No other flags are affected. CLR can operate on the carry or any directly addressable bit. Note : When this instruction is used to modify an output pin, the value used as the original data will be read from the output data latch, not the input pin. | | |
| Example : | Port 1 has previously been written with FFH (11111111B). The instruction CPL P1.1 will leave the port set to FDH (11111101B) | | |
| CPL C | | | |
| Operation : | $(C) \leftarrow \overline{(C)}$ | | |
| CPL bit | | | |
| Operation : | $(bit) \leftarrow \overline{(bit)}$ | | |
| <hr/> | | | |
| ANL C,<src-bit> | Function : Logical-AND for bit variables | Bytes : 2 | Cycles : 2 |
| Description : | If the Boolean value of the source bit is a logical 0 then clear the carry flag; otherwise leave the carry flag in its current state. A slash ("/") preceding the operand in the assembly language indicates that the logical complement of the addressed bit is used as the source value, but the source bit itself is not affected. No flags are affected. Only direct addressing is allowed for the source operand. | | |
| Example : | Set the carry flag if, and only if, P1.0 = 1, ACC.7 = 1, and OV = 0: MOV C, P1.0 ; LOAD CARRY WITH INPUT PIN STATE ANL C, ACC.7 ; AND CARRY WITH ACCUMULATOR BIT 7 ANL C,/OV ; AND WITH INVERSE OF OVERFLOW FLAG | | |
| ANL C, bit | | | |
| Operation : | $(C) \leftarrow (C) \wedge (bit)$ | | |
| ANL C/ bit | | | |

Operation : $(C) \leftarrow (C) \wedge \overline{(\text{bit})}$

ORL C, <src-bit>**Bytes : 2 Cycles : 2****Function : Logical-OR for bit variables**

Description : Set the carry flag if the Boolean value is a logical 1; leave the carry in its current state otherwise. Slash ("/") preceding the operand in the assembly language indicates that the logical complement of the addressed bit is used as the source value, but the source bit itself is not affected. No other flags are affected.

Example : Set the carry flag if and only if $P1.0 = 1$, $ACC.7 = 1$, or $OV = 0$:

```
MOV C,P1.0 ; LOAD CARRY WITH INPUT PIN P10
ORL C,ACC.7 ; OR CARRY WITH THE ACC.BIT7
ORL C,/OV ; OR CARRY WITH THE INVERSE OF OV.
```

ORL C, bit**Operation :** $(C) \leftarrow (C) \vee (\text{bit})$ **ORL C,/bit****Operation :** $(C) \leftarrow (C) \vee \overline{(\text{bit})}$

MOV <dest-bit>, <src-bit> Function : Move bit data Bytes : 2 Cycles : 1/2

Description : The Boolean variable indicated by the second operand is copied into the location specified by the first operand. One of the operands must be the carry flag; the other may be any directly addressable bit. No other register or flag is affected.

Example : The carry flag is originally set. The data present at input Port 3 is (11000101B). The data previously written to output Port 1 is C5H (00110101B).

```
MOV P1.3, C
```

```
MOV C, P3.3
```

MOV P1.2, C will leave the carry cleared and change Port 1 to 39H (00111001B).

MOV C, bit**Operation :** $(C) \leftarrow (\text{bit})$

MOV bit, COperation : $(C) \leftarrow (\text{bit})$ **8.3.3 Rotate and Swap Operations**

The Table 8.11 gives the list of rotate and swap operations supported by 8051.

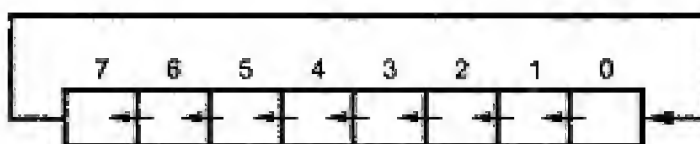
| Mnemonic | Operand | Destination | Byte | Cyc |
|----------|---------|--------------------------------------|------|-----|
| RL | A | Rotate Accumulator left | 1 | 1 |
| RLC | A | Rotate A left through the Carry flag | 1 | 1 |
| RR | A | Rotate Accumulator Right | 1 | 1 |
| RRC | A | Rotate A Right through Carry flag | 1 | 1 |
| SWAP | A | Swap nibbles within the Accumulator | 1 | 1 |

Table 8.11 List of rotate and swap operations

Rotate and Swap Instructions with Examples

| | | | |
|-------------|---|------------------|-------------------|
| RL A | Function : Rotate Accumulator Left | Bytes : 1 | Cycles : 1 |
|-------------|---|------------------|-------------------|

Description : The eight bits in the Accumulator are rotated one bit to the left. Bit 7 is rotated into the bit 0 position. No flags are affected.

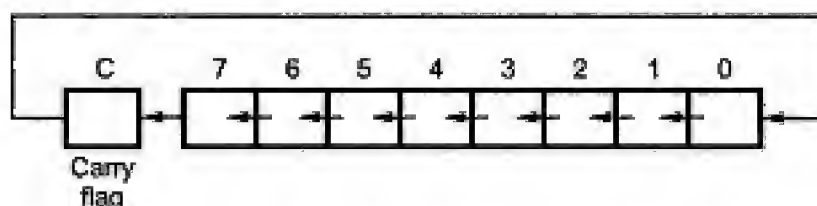


Example : The Accumulator holds are value C5H (11000101B). The instruction, RL, A leaves the Accumulator holding the value 8BH (10001011B) with the carry unaffected.

Operation : $(A_{n+1}) \leftarrow (A_n) \quad n = 0 - 6$
 $(A_0) \leftarrow (A_7)$

RLC A Bytes : 1 Cycles : 1 Function : Rotate Accumulator Left through the Carry flag

Description : The eight bits in the Accumulator and the carry flag are together rotated one bit to the left. Bit 7 moves into the carry flag; the original state of the carry flag moves into the bit 0 position. No other flags are affected.



Example : The Accumulator holds the value C5H (11000101B), and the carry is zero. The instruction,

RLC A

leaves the Accumulator holding the value 8BH (10001010B) with the carry set.

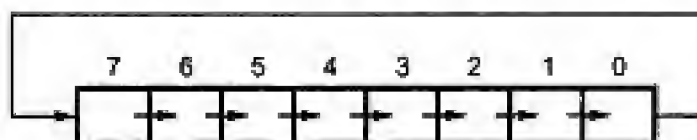
Operation : $(A_{n+1}) \leftarrow (A_n) \quad n = 0 - 6$

$(A_0) \leftarrow (C)$

$(C) \leftarrow (A_7)$

RR A Function : Rotate Accumulator Right Bytes : 1 Cycles : 1

Description : The eight bits in the Accumulator are rotated one bit to the right. Bit 0 is rotated into the bit 7 position. No flags are affected.



Example : The Accumulator holds the value C5H (11000101B) The instruction,
RR A

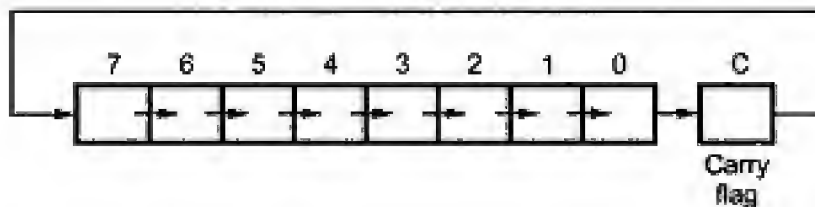
leaves the Accumulator holding the value E2H (11100010B) with the carry unaffected.

Operation : $(A_n) \leftarrow (A_{n+1}) \quad n = 0 - 6$

$(A_7) \leftarrow (A_0)$

RRC A Bytes : 1 Cycles : 1 Function : Rotate Accumulator Right through Carry flag

Description : The eight bits in the Accumulator and the carry flag are together rotated one bit to the right. Bit 0 moves into the carry flag; the original value of the carry flag moves into the bit 7 position. No other flags are affected.



Example : The Accumulator holds the value C5H (11000101B), the carry is zero. The instruction

RRC A

leaves the Accumulator holding the value 62 (01100010B) with the carry set.

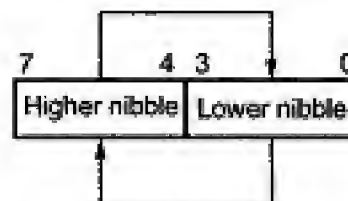
Operation : $(A_n) \leftarrow (A_n + 1) \quad n = 0 - 6$

$(A_7) \leftarrow (C)$

$(C) \leftarrow (A_0)$

SWAP A Function : Swap nibbles within the Accumulator Bytes : 1 Cycles : 1

Description : Swap A interchanges the low-and high-order nibbles (four-bit fields) of the Accumulator (bits 3-0 and bits 7-4). The operation can also be through of as a four-bit rotate instruction. No flags are affected.



Example : The Accumulator holds the value C5H (11000101B). The instruction

SWAP A

leaves the Accumulator holding the value 5CH (01011100B)

Operation : SWAP

$$(A_{3-0}) \xrightarrow{\quad} (A_{7-4})$$

$$\quad \quad \quad \leftarrow$$

8.4 Arithmetic Instructions

The arithmetic operations of 8051 include increment, decrement, addition, subtraction, multiplication, division and decimal operations. Like logical operations, arithmetic operations also affect flags. Let us see the instructions which affect flags before going to study arithmetic instructions in detail.

8.4.1 Flags

We know that, the 8051 has four flags : the Carry (C), Auxiliary Carry (AC), Overflow (OV), and Parity (P). The C, AC and OV flags are arithmetic flags. They are set to 1 or cleared to 0 automatically, depending on the result of instructions, shown in Table 8.12.

| Instruction | Flags Affected | | |
|---------------|--------------------|----|----|
| | C | AC | OV |
| ADD | C | AC | OV |
| ADDC | C | AC | OV |
| ANL C, direct | C | | |
| CJNE | C | | |
| CLR C | C = 0 | | |
| CPL C | C = \overline{C} | | |
| DAA | C | | |
| DIV | C = 0 | OV | |
| MOV C, direct | C | | |
| MUL | C = 0 | OV | |
| ORL C, direct | C | | |
| RLC | C | | |
| RRC | C | | |
| SETB C | C = 1 | | |
| SUBB | C | AC | OV |

Table 8.12 Flags affected by instructions

8.4.2 Incrementing and Decrementing

Incrementing and decrementing instructions allow addition and subtraction of 1 from a given number. These instructions not affect C, AC and OV flags. The Table 8.13 lists the increment and decrement instructions.

| Mnemonic | Operand | Description | Byte | Cyc |
|----------|---------|------------------------|------|-----|
| INC | A | Increment Accumulator | 1 | 1 |
| INC | Rn | Increment register | 1 | 1 |
| INC | direct | Increment direct byte | 2 | 1 |
| INC | @Ri | Increment indirect RAM | 1 | 1 |
| DEC | A | Decrement Accumulator | 1 | 1 |
| DEC | Rn | Decrement register | 1 | 1 |
| DEC | direct | Decrement direct byte | 2 | 1 |
| DEC | @Ri | Decrement indirect RAM | 1 | 1 |
| INC | DPTR | Increment data Pointer | 1 | 2 |

Table 8.13

Incrementing and Decrementing Instructions

| INC <byte> | Function : Increment | Bytes : 1/2 | Cycles : 1 |
|----------------------|--|-------------|------------|
| Description : | INC increments the indicated variable by 1. An original value of 0FFH will overflow to 00H. No flags are affected. Three addressing modes are allowed: Register, direct or register-indirect. Note : When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, not the input pins. | | |
| Example : | Register 0 contains 7EH (011111110B). Internal RAM locations 7EH and 7FH contain 0FFH and 40H, respectively. The instruction sequence, INC @R0 INC R0 INC @R0 will leave register 0 set to 7FH and internal RAM locations 7EH and 7FH holding (respectively) 00H and 41H. | | |

INC A

Operation : $(A) \leftarrow (A) + 1$

INC Rn

Operation : $(Rn) \leftarrow (Rn) + 1$

Example : INC R2 ; increments contents of R2 by 1

INC direct

Operation : $(\text{direct}) \leftarrow (\text{direct}) + 1$

Example : INC 20H ; increments contents of memory location whose address is given within the instruction (20H).

INC @Ri

Operation : $((Ri)) \leftarrow ((Ri)) + 1$

Example : INC @R2 ; increment contents of memory location whose address is given by register R2 by 1.

| INC DPTR | Function : Increment Data Pointer | Bytes : 1 | Cycles : 2 |
|----------------------|--|------------------|-------------------|
| Description : | Increment the 16-bit data pointer by 1. A 16-bit increment (modulo 2^{16}) is performed; an overflow of the low-order byte of the data pointer (DPL) from FFH to 00H will increment the high-order byte (DPH). No flags are affected. This is the only 16-bit register which can be incremented. | | |
| Example : | Registers DPH and DPL contain 12H and FEH, respectively. The instruction sequence, INC DPTR INC DPTR INC DPTR will change DPH and DPL to 13H and 01H. | | |
| Operation : | $(DPTR) \leftarrow (DPTR) + 1$ | | |

| DEC byte | Function : Decrement | Bytes : 1/2 | Cycles : 1 |
|----------------------|--|-------------|------------|
| Description : | The variable indicated is decremented by 1. An original value of 00H will underflow to 0FFH. No flags are affected. Four operand addressing modes are allowed: accumulator, register, direct, or register-indirect. Note : When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, not the input pins. | | |
| Example : | Register 0 contains 7FH (01111111B). Internal RAM locations 7EH and 7FH contain 00H and 40H, respectively. The instruction sequence, DEC @R0 DEC R0 DEC @R0 will leave register 0 set to 7EH and internal RAM locations 7EH and 7FH set to 0FFH and 3FH. | | |
| DEC A | | | |
| Operation : | $(A) \leftarrow (A) - 1$ | | |
| DEC Rn | | | |
| Operation : | $(Rn) \leftarrow (Rn) - 1$ | | |
| Example : | DEC R3 ; Decrements contents of R3 by 1 | | |
| DEC direct | | | |
| Operation : | $(\text{direct}) \leftarrow (\text{direct}) - 1$ | | |
| Example : | DEC 20H ; Decrements the contents of memory location whose address is 20H by 1. | | |
| DEC @Rn | | | |
| Operation : | $((Ri)) \leftarrow ((Ri)) - 1$ | | |
| Example : | DEC @R2 ; Decrements the contents of memory location whose address is given by register R2 by 1. | | |

8.4.3 Addition

The 8051 supports two addition instructions : ADD and ADDC. The ADD instruction adds a byte variable with the accumulator, leaving the result in the accumulator. The carry flag is set if there is an overflow from bit 7 and cleared otherwise. The AC flag is set to the carry-out from bit 3 for use by the DA instruction described later. ADDC adds the previous contents of the carry flag with the two byte variables, but otherwise is the same as ADD.

The Table shows the list of addition instructions supported by 8051.

| Mnemonic | Operand | Description | Byte | Cyc |
|----------|----------|---|------|-----|
| ADD | A,Rn | Add register to Accumulator | 1 | 1 |
| ADD | A,direct | Add direct byte to Accumulator | 2 | 1 |
| ADD | A,@Ri | Add indirect RAM to Accumulator | 1 | 1 |
| ADD | A,#data | Add immediate data to Accumulator | 2 | 1 |
| ADDC | A,Rn | Add register to Accumulator with Carry | 1 | 1 |
| ADDC | A,direct | Add direct byte to A with carry flag | 2 | 1 |
| ADDC | A,@Ri | Add indirect RAM to A with Carry flag | 1 | 1 |
| ADDC | A,#data | Add immediate data to A with Carry flag | 2 | 1 |

Addition Instructions

| ADD A,<src-byte> | Function : Add | Bytes : 1/2 | Cycles : 1 |
|------------------|----------------|-------------|------------|
|------------------|----------------|-------------|------------|

Description : ADD adds the byte variable indicated to the Accumulator, leaving the result in the Accumulator. The carry and auxiliary-carry flags are set, respectively, if there is a carry-out from bit 7 or bit 3, and cleared otherwise. When adding unsigned integers, the carry flag indicates an overflow occurred.

When adding signed integers, OV indicates a negative number produced as the sum of two positive operands, or a positive sum from two negative operands.

Four source operand addressing modes are allowed: register, direct, register-indirect, or immediate.

Example : The Accumulator holds 89H (10001001B) and register 0 holds 95H (10010101B). The instruction

ADD A, R0

will leave 1E (00011110B) in the Accumulator with the AC flag cleared and both the carry flag and OV set to 1.

ADD A, Rn

Operation : $(A) \leftarrow (A) + (Rn)$

Example : **ADD A, R2 ;** Adds contents of A and R2 and store result in A.

ADD A, direct

Operation : $(A) \leftarrow (A) + (\text{direct})$

Example : **ADD A, 20H ;** Adds contents of A and memory whose address is 20H, and store result in A.

ADD A, @Ri

Operation : $(A) \leftarrow (A) + ((Ri))$

Example : **ADD A, @R2;** Adds contents of A and memory whose address is given by register R2, and store result in A

ADD A, #data

Operation : $(A) \leftarrow (A) + \#data$

Example : **ADD A, #20H ;** Adds the contents of A and 20H

ADDC A, <src-byte> Function : Add with Carry Bytes : 1/2 Cycles : 1

Description : ADC simultaneously adds the byte variable indicated, the carry flag and the Accumulator respectively, if there is a carry-out from bit 7 or bit 3, and cleared otherwise. When adding unsigned integers, the carry flag indicates an overflow occurred.

When adding signed integers, OV indicates a negative number produced as the sum of two positive operands or a positive sum from two negative operands.

Four source operand addressing modes are allowed : register-indirect, or immediate.

Example : The Accumulator holds 89H (10001001B) and register 0 holds 95H (10010101B) with the carry flag set. The instruction,

ADDC ..., R0

will leave 1FH (00011111B) in the Accumulator with AC cleared and both the Carry flag and OV set to 1.

ADDC A, Rn

Operation : $(A) \leftarrow (A) + (C) + (Rn)$

Example : ADDC A, R2 ; Adds the contents of A, R2 and carry flag, and stored result in A.

ADDC A, direct

Operation : $(A) \leftarrow (A) + (C) + (\text{direct})$

Example : ADDC A, 20H ; Adds the contents of A, memory location whose address is 20H and the carry flag and stores result in A

ADDC A, @Ri

Operation : $(A) \leftarrow (A) + (C) + ((R_i))$

Example : ADDC A, @R2 ; Adds the contents of A, memory location whose address is given by register R2 and the carry flag, and stores result in the A.

ADDC A, #data

Operation : $(A) \leftarrow (A) + (C) + \#data$

Example : ADDC A, #20H ; Adds the contents of A and carry flag and 20H and stores result in A.

8.4.4 Subtraction

The 8051 supports one subtraction instruction SUBB. The SUBB (subtract with borrow) instruction subtracts the byte variable indicated and the contents of the carry flag together from the accumulator, and puts the result back in the accumulator. The carry flag serves as a "Borrow Required" flag during subtraction operations when a greater value is subtracted from a lesser value (as in subtracting 5 from 1) requiring a borrow into the highest order bit, the carry flag is set; otherwise it is cleared.

The Table shows the list of subtraction instructions supported by 8051.

| Mnemonic | Operand | Description | Byte | Cyc |
|----------|----------|--|------|-----|
| SUBB | A,Rn | Subtract register from A with Borrow | 1 | 1 |
| SUBB | A,direct | Subtract direct byte from A with Borrow | 2 | 1 |
| SUBB | A,@Ri | Subtract indirect RAM from A with Borrow | 1 | 1 |
| SUBB | A,#data | Subtract immediate data from A with Borrow | 2 | 1 |

Subtraction Instructions

SUBB A, <src-byte> Function : Subtract with borrow Bytes : 2 Cycles : 1

Description : SUBB subtracts the indicated variable and the carry flag together from the Accumulator, leaving the result in the Accumulator. SUBB sets the carry (borrow) flag if a borrow is needed for bit 7, and clears otherwise. (If C was set before executing a SUBB instruction, this indicates that a borrow was needed for the previous step in a multiple precision subtraction, so the carry is subtracted from the Accumulator along with the source operand.) AC is set if a borrow is needed for bit 3, and cleared otherwise. OV is set if a borrow is needed into bit 6, but not into bit 7, or into bit 7, but not bit 6.

When subtracting signed integers OV indicates a negative number produced when a negative value is subtracted from a positive value, or a positive result when a positive number is subtracted from a negative number.

The source operand allows four addressing modes: register, direct, register-indirect, or immediate.

Example : The Accumulator holds C9H (11001001B), register, 2 holds 54H (01010100B), and the carry flag is set. The instruction,

SUBB A, R2

will leave the value 74H (01110100B) in the accumulator, with the carry flag and AC cleared but OV set.

Notice that C9H minus 54H is 75H. The difference between this and the above result is due to the carry(borrow) flag being set before the operation. If the state of the carry is not known before starting a single or multiple-precision subtraction, it should be explicitly cleared by a CLR C instruction.

SUBB A, Rn

Operation : $(A) \leftarrow (A) - (C) - (Rn)$

Example : SUBB A, R3 ; Subtracts contents of R3 and carry together from A and stores results in A.

SUBB A, direct

Operation : $(A) \leftarrow (A) - (C) - (\text{direct})$

Example : SUBB A, 20H ; subtracts the contents of memory location 20H and carry together from A stores result in A.

SUBB A, @Ri

Operation : $(A) \leftarrow (A) - (C) - ((Ri))$

Example : SUBB A, @R2 ; Subtracts the contents of memory location whose address is given by R2 and carry together from A and stores result in A.

SUBB A, #data

Operation : $(A) \leftarrow (A) - (C) - \#data$

Example : SUBB A, #20H ; Subtracts 20H from A and stores result in A.

8.4.5 Multiplication and Division

The instruction "MUL AB" multiplies the unsigned eight-bit integer values held in the accumulator and B-register. The low-order byte of the sixteen-bit product is left in the accumulator, the higher-order byte in B. If the high-order eight-bits of the product are all zero, the overflow flag is cleared; otherwise it is set. The programmer can check OV to determine when the B register is non-zero and must be processed.

"DIV AB" divides the unsigned eight-bit integer in the accumulator by the unsigned eight-bit integer in the B-register. The integer part of the quotient is returned in the accumulator, the remainder in the B register. If the B-register originally contains 00H then the overflow flag will be set to indicate a division error, and the values returned will be undefined. Otherwise OV is cleared.

The divide instruction is also useful for purposes such as radix conversion or separating bit fields of the accumulator. A short subroutine can convert an eight-bit unsigned binary integer in the accumulator (between 0 and 255) to a three-digit (two byte) BCD representation. The hundred's digit is returned in one register (HUND) and the ten's and one's digits returned as packed BCD in another (TENONE).

➡ **Example 1 :** Use of DIV instruction for Radix Conversion.

```

BINBCD:      CONVERT B-BIT BINARY VARIABLE IN ACC
              TO 3-DIGIT PACKED BCD FORMAT
              HUNDREDS PLACE LEFT IN VARIABLE 'HUND'
              TENS AND ONES PLACES IN 'TENONE'
HUND EQU     21H
TENONE EQU   22H
BINBCD:      MOV     B, #100      ; DIVIDE BY 100 TO
              DIV     AB          ; DETERMINE NUMBER OF HUNDREDS
              MOV     HUND, A
              MOV     A, #10      ; LOAD DIVIDER
              XCH     A, B        ; COUNT
              DIV     AB          ; DIVIDE
              ; REMAINDER BY 10

```

```

        SWAP      A
        ADD       A,B          ; PACK BCD DIGITS IN ACC
        MOV      TENONE,A
        RET

```

The divide instruction can also separate eight bits of data in the accumulator into sub-fields. For example, packed BCD data may be separated into two nibbles by dividing the data by 16 leaving the high-nibble in the accumulator and the low-order nibble (remainder) in B. The two digits may then be operated on individually or in conjunction with each other. This example receives two packed BCD digits in the accumulator and returns the product of the two individual digits in packed BCD format in the accumulator.

➡ **Example 2 : Implementing a BCD Multiply Using MPY and DIV.**

```

MULBCD:  UNPACK TWO BCD DIGITS RECEIVED IN ACC,
        FIND THEIR PRODUCT,
        AND RETURN PRODUCT IN PACKED BCD FORMAT IN ACC
MULBCD:  MOV      B,#10H      ; DIVIDE INPUT BY 10
        DIV      AB          ; A AND B HOLD SEPARATED DIGITS
        ; (EACH RIGHT JUSTIFIED IN REGISTER)
        MUL      AB          ; A HOLDS PRODUCT IN
        ; BINARY FORMAT
        ; (0 - 99 (DECIMAL) = 0 - 83H)
        MOV      B,#10       ; DIVIDE PRODUCT BY 10
        DIV      AB          ; A HOLDS # OF TENS, B
        ; HOLDS REMAINDER
        SWAP     A
        ORL      A,B         ; PACK DIGITS
        RET

```

Multiplication Instruction

| MUL AB | Function : Multiply | Bytes : 1 | Cycles : 4 |
|----------------------|--|-----------|------------|
| Description : | MUL AB multiplies the unsigned eight-bit integers in the Accumulator and register B. The low-order byte of the sixteen-bit product is left in the Accumulator, and the high-order byte in B. If the product is greater than 255 (FFH) the overflow flag is set; otherwise it is cleared. The carry flag is always cleared. | | |
| Example : | Originally the Accumulator holds the value 80 (50H). Register B holds the value 160 (0A0H). The instruction, MUL AB will give the product 12,800 (3200H), so B is changed to 32H (00110010B) and the Accumulator is cleared. The overflow flag is set, carry is cleared. | | |
| Operation : | $(A)_{7-0} \leftarrow (A) \times (B)$ $(B)_{15-8}$ | | |

Division Instruction

| DIV AB | Function : Divide | Bytes : 1 | Cycles : 4 |
|----------------------|--|-----------|------------|
| Description : | <p>DIV AB divides the unsigned eight-bit integer in the Accumulator by the unsigned eight-bit integer in register B. The Accumulator receives the integer part of the quotient; register B receives the integer remainder. The carry and OV flags will be cleared.</p> <p>Exception : If B had originally contained 00H, the values returned in the Accumulator and B-register will be undefined and the overflow flag will be set. The carry flag is cleared in any case.</p> | | |
| Example : | <p>The Accumulator contains 250 (0FBH or 11111010B) and B contains 18 (12H or 00010010B). The instruction,</p> <p>DIV AB</p> <p>will leave 13 in the Accumulator (0DH or 00001101B) and the value 16 (10H or 00010000B) in B, since $250 = (13 \times 18) + 16$. Carry and OV will both be cleared.</p> | | |
| Operation : | <p>DIV</p> $(A)_{15-8} \leftarrow (A)/(B)$ $(B)_{7-0}$ | | |

8.4.6 Decimal Arithmetic

Decimal addition is possible by using the DA instruction in conjunction with ADD and/or ADDC. The eight-bit binary value in the accumulator resulting from an earlier addition of two variables (each a packed BCD digit-pair) is adjusted to form two BCD digits of four bits each. If the contents of accumulator bits 3-0 are greater than nine or if the AC flag had been set, six is added to accumulator producing the proper BCD digit in the low-order nibble. (This addition might itself set - but would not clear- the carry flag.) If the carry flag is set, or if the four high-order bits now exceed nine these bits are incremented by six. The carry flag is left set if originally set or if either addition of six produces a carry out of the highest-order bit, indicating the sum of the original two BCD variables is greater than or equal to decimal 100.

➡ **Example 3 :** Two Byte Decimal Add with Registers and Constants.

```
BCDADD:      ADD THE CONSTANT 1,234 (DECIMAL) TO
              THE CONTENTS OF REGISTER PAIR <R3><R2>
BCDADD      MOV      A,R2
              ADD      A,# 34
              DA        A
              MOV      R2,A
```

| | |
|------|----------|
| MOV | A, R3 |
| ADDC | A, # 12H |
| DA | A |
| MOV | R3, A |
| RET | |

DA A Function : Decimal-adjust Accumulator for addition Bytes : 1 Cycles : 1

Description :

DA A adjusts the eight-bit value in the Accumulator resulting from the earlier addition of two variables (each in packed-BCD format), producing two four-bit digits. Any ADD or ADC instruction may have been used to perform the addition.

If Accumulator bits 3-0 are greater than nine (xxxx1010-xxxx1111), or if the AC flag is one, six is added to the Accumulator producing the proper BCD digit in the low-order nibble. This internal addition would set the carry flag if a carry-out of the low-order four-bit field propagated through all high order bits, but it would not clear the carry flag otherwise.

If the carry flag is now set, or if the four high-order bits now exceed nine (1010xxxx-1111xxxx), these high-order bits are incremented by six, producing the proper BCD digit in the high-order nibble. Again, this would set the carry flag if there was a carry-out of the high-order bits, but wouldn't clear the carry. The carry flag thus indicates if the sum of the original two BCD variables is greater than 100, allowing multiple precision decimal addition. OV is not affected.

All of this occurs during the one instruction cycle. Essentially, this instruction performs the decimal conversion by adding 00H, 06H, 60H, or 66H to the Accumulator, depending on initial Accumulator and PSW conditions.

Note : DA A cannot simply convert a hexadecimal number in the Accumulator to BCD notation, nor does DA A apply to decimal subtraction.

Example :

The Accumulator holds the value 55H (01010101B) representing the packed BCD digits of the decimal number 55. Register 3 contains the value 68H (01101000B) representing the packed BCD digits of the decimal number 68. The carry flag is set. The instruction sequence.

```
ADDC A, R3
```

```
DA    A
```

will first perform a standard two's-complement binary addition, resulting in the value BEH (10111110) in the Accumulator. The carry and auxiliary carry flags will be cleared.

The Decimal Adjust instruction will then alter the Accumulator to the value 24H (00100100B), indicating the packed BCD digits of the decimal number 24, the low-order two digits of the decimal sum of 55, 68, and the carry-in. The carry flag will be set by the Decimal Adjust instruction, indicating that a decimal overflow occurred. The true sum 55, 68 and 1 is 124.

BCD variables can be incremented or decremented by adding 01H or 99H. If the Accumulator initially holds 30H (representing the digits of 30 decimal), then the instruction sequence.

```
ADD A, # 99 H
```

```
DA A
```

will leave the carry set and 29H in the Accumulator, since $30 + 99 = 129$. The low-order byte of the sum can be interpreted to mean $30 - 1 = 29$.

Operation :

DA

```
IF [ [(A3-0) > 9] OR [(AC) = 1] ]
    THEN (A3-0) ← (A3-0) + 6
    AND
IF [ [(A7-4) > 9] OR [(C) = 1] ]
    THEN (A7-4) ← (A7-4) + 6
```

8.5 Jump and CALL Instructions

Jump and CALL instructions change the flow of the program by changing the contents of program counter. A jump permanently changes the program flow whereas call temporarily changes the program flow to allow another part of the program to run. There are jump instructions which change the program flow if certain condition exists. For example, CJNE (compare and jump if not equal). This instruction compares the magnitude of the first two operands, and changes program flow if their values are not equal.

The following types of instructions change the program flow :

- Jump on bit conditions.
- Compare byte and jump if not equal.
- Decrement byte and jump if zero.
- Jump unconditionally.
- Call a subroutine.
- Return from a subroutine.

8.5.1 Jump and Call Program Range

We know that a jump and call instructions replace the contents of the program counter with a new program address. The new address can be specified either by specifying the difference between the new address and the current program counter contents or by specifying the entire new address. The difference, in bytes, of the new address from the address in the program counter is called the range of the jump or call. For example, if a jump instruction is located at program address 0200H, and the jump causes the program counter to become 0230H, then the range of the jump is 30H bytes. Jump and CALL instructions may have one of the three ranges.

- Relative range : +127 to - 128 (+7FH to - 80H)
- Absolute range : 0000H to 07FFH
- Long range : 0000H to FFFFH

8.5.2 Jump

The 8051 has three forms of jump instruction. Each causes the program to unconditionally jump to some other address. They differ in how the machine code represents the destination address.

LJMP (Long Jump) encodes a sixteen-bit address in the second and third instruction bytes (Fig. 8.6); the destination may be anywhere in the 64 KiloByte program memory address space.

The two byte AJMP (Absolute Jump) instruction encodes its destination using address bits 10 through 8 form a three bit field in the opcode and address bits 7 through 0 form the second byte. Address bits 15-12 are unchanged from the (incremented) contents of the PC, so AJMP can only be used when the destination is known to be within the same 2K memory block (otherwise 8051 will point out the error).

a) Long jump (LJMP addr 16) :



b) Absolute jump (AJMP addr 11) :



c) Short jump (SJMP rel) :



Fig. 8.6 Jump instruction machine code formats

A different two-byte jump instruction is legal with any nearby destination, regardless of memory block boundaries or "pages" SJMP (Short Jump) encodes the destination with a program counter-relative address in the second byte. The CPU calculates the destination at run-time by adding the signed eight-bit displacement value to the incremented PC. Negative offset values will cause jumps upto 128 bytes backwards; positive values up to 127 bytes forwards. (SJMP with 00H in the machine code offset byte will proceed with the following instruction).

Like SJMP all conditional jump instructions use relative addressing. JZ (Jump if Zero) and JNZ (Jump if Not Zero) monitor the state of the accumulator as implied by their names, while JC (Jump on Carry) and JNC (Jump on No Carry) test whether or not the carry flag is set. All four are two-byte instructions. JB (Jump on Bit), JNB (Jump on No Bit) and JBC (Jump on Bit then Clear Bit) can test any status bit or input pin with a three byte instruction; the second byte specifies which bit to test and the third gives the relative offset value.

The Table 8.14 shows the list of jump instructions supported by 8051.

| Mnemonic | Operand | Description | Byte | Cyc |
|----------|------------------------------|---|------|-----|
| AJMP | addr11 | Absolute Jump | 2 | 2 |
| LJMP | addr16 | Long Jump | 3 | 2 |
| SJMP | rel | Short Jump (relative addr) | 2 | 2 |
| JMP | @A+DPTR | Jump indirect relative to the DPTR | 1 | 2 |
| JZ | rel | Jump if Accumulator is Zero | 2 | 2 |
| JNZ | rel | Jump if Accumulator is Not Zero | 2 | 2 |
| JC | rel | Jump if Carry flag is set | 2 | 2 |
| JNC | rel | Jump if No Carry flag | 2 | 2 |
| JB | bit, rel | Jump if direct Bit set | 3 | 2 |
| JNB | bit, rel | Jump if direct Bit Not set | 3 | 2 |
| JBC | bit, rel | Jump if direct Bit is set and Clear bit | 3 | 2 |
| CJNE | <dest-byte>, <src-byte>, rel | Compare and Jump if Not Equal | 3 | 2 |
| DJNZ | byte, <rel-addr> | Decrement and Jump if Not Zero | 3/2 | 2 |

Table 8.14

Jump Instructions

| AJMP addr11 | Function : Absolute Jump | Bytes : 2 | Cycles : 2 |
|-------------|--------------------------|-----------|------------|
|-------------|--------------------------|-----------|------------|

Description :

AJMP transfers program execution to the indicated address, which is formed at run-time by concatenating the high-order five bits of the PC (after incrementing the PC twice), opcode bits 7-5, and the second byte of the instruction. The destination must therefore be

within the same 2K block of program memory as the first byte of the instruction following AJMP.

Example : The label "JMPADR" is at program memory location 0125H. The instruction

AJMP JMPADR

is at location 0345H and will load the PC with 0125H.

Operation : AJMP

$(PC) \leftarrow (PC) + 2$

$(PC_{10-0}) \leftarrow \text{Page address}$

LJMP addr16**Function : Long Jump****Bytes : 3****Cycles : 2**

Description : LJMP causes an unconditional branch to the indicated address by loading the high-order and low-order bytes of the PC(respectively) with the second and third instruction bytes. The destination may therefore be anywhere in the full 64K program memory address space. No flags are affected.

Example : The label "JMPADR" is assigned to the instruction at program memory location 1234H. The instruction,

LJMP JMPADR

at location 0123H will load the program counter with 1234H.

Operation : LJMP

$(PC) \leftarrow \text{addr}_{15-0}$

SJMP rel**Function : Short Jump****Bytes : 2****Cycles : 2**

Description : Program control branches unconditionally to the address indicated. The branch destination is computed by adding the signed displacement in the second instruction byte to the PC, after incrementing the PC twice. Therefore, the range of destinations allowed is from 128 bytes preceding this instruction to 127 bytes following it.

Example : The label "RELADR" is assigned to an instruction at program memory location 0123H. The instruction,

SJMP RELADR

will assemble into location 0100H. After the instruction is executed, the PC will contain the value 0123H.

(Note : Under the above conditions the instruction following SJMP will be at 102H. Therefore, the displacement byte of the instruction will be the relative offset (0123H-0102H) = 21H. Put another way, an SJMP with a displacement of 0FEH would be a one-instruction infinite loop.)

Operation : SJMP
 $(PC) \leftarrow (PC) + 2$
 $(PC) \leftarrow (PC) + rel$

| | | | |
|----------------------|---------------------------------|------------------|-------------------|
| JMP @A + DPTR | Function : Jump indirect | Bytes : 1 | Cycles : 2 |
|----------------------|---------------------------------|------------------|-------------------|

Description : Add the eight-bit unsigned contents of the Accumulator with the sixteen-bit data pointer, and load the resulting sum to the program counter. This will be the address for subsequent instruction fetches. Sixteen-bit addition is performed (modulo 2^{16}); a carry-out from the low-order eight bits propagates through the higher-order bits. Neither the Accumulator nor the Data Pointer is altered. No flags are affected.

Example : An even number from 0 to 6 is in the Accumulator. The following sequence of instructions will branch to one of four AJMP instructions in a jump table starting at JMP_TBL :

```

                                MOV        DPTR, #JMP_TBL
                                JMP        @A+DPTR
JMP_TBL :  AJMP        LABEL0
                                AJMP        LABEL1
                                AJMP        LABEL2
                                AJMP        LABEL3

```

If the Accumulator equals 04H when starting this sequence, execution will jump to label LABEL2. Remember that AJMP is a two-byte instruction, so the jump instructions start at every other address.

Operation : $(PC) \leftarrow (A) + (DPTR)$

| JZ rel | Function : Jump if Accumulator Zero | Bytes : 2 | Cycles : 2 |
|----------------------|---|------------------|-------------------|
| Description : | If all bits of the Accumulator are zero, branch to the address indicated; otherwise proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the second instruction byte to the PC, after incrementing the PC twice. The Accumulator is not modified. No flags are affected. | | |
| Example : | <p>The Accumulator originally contains 01H. The instruction sequence,</p> <pre>JZ LABEL1 DEC A JZ LABEL2</pre> <p>will change the Accumulator to 00H and cause program execution to continue at the instruction identified by the label LABEL2.</p> | | |
| Operation : | $(PC) \leftarrow (PC) + 2$ IF (A) = 0 THEN $(PC) \leftarrow (PC) + rel$ | | |

| JNZ rel | Function : Jump if Accumulator Not Zero | Bytes : 2 | Cycles : 2 |
|----------------------|--|------------------|-------------------|
| Description : | If any bit of the Accumulator is a one, branch to the indicated address; otherwise proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the second instruction byte to the PC, after incrementing the PC twice. The Accumulator is not modified. No flags are affected. | | |
| Example : | <p>The Accumulator originally hold 00H. The instruction sequence,</p> <pre>JNZ LABEL INC A JNZ LABEL2</pre> <p>will set the Accumulator to 01H and continue at label LABEL2.</p> | | |
| Operation : | $(PC) \leftarrow (PC) + 2$ IF (A) \neq 0 THEN $(PC) \leftarrow (PC) + rel$ | | |

| JC rel | Function : Jump if Carry is set | Bytes : 2 | Cycles : 2 |
|----------------------|--|-----------|------------|
| Description : | If the carry flag is set branch to the address indicated; otherwise proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the second instruction byte to the PC, after incrementing the PC twice. No flags are affected. | | |
| Example : | <p>The carry flag is cleared. The instruction sequence,</p> <pre>JC LABEL 1 CPL C JC LABEL 2</pre> <p>will set the carry and cause program execution to continue at the instruction identified by the label LABEL2.</p> | | |
| Operation : | $(PC) \leftarrow (PC) + 2$ IF (C) = 1 THEN $(PC) \leftarrow (PC) + rel$ | | |

| JNC rel | Function : Jump if Carry not set | Bytes : 2 | Cycles : 2 |
|----------------------|--|-----------|------------|
| Description : | If the carry flag is a zero, branch to the address indicated; otherwise proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the second instruction byte to the PC, after incrementing the PC twice to point to the next instruction. The carry flag is not modified. | | |
| Example : | <p>The carry flag is set. The instruction sequence,</p> <pre>JNC LABEL1 CPL C JNC LABEL2</pre> <p>will clear the carry and cause program execution to continue at the instruction identified by the label LABEL2.</p> | | |
| Operation : | $(PC) \leftarrow (PC) + 2$ IF (C) = 0 THEN $(PC) \leftarrow (PC) + rel$ | | |

| JB bit, rel | Function : Jump if Bit set | Bytes : 3 | Cycles : 2 |
|----------------------|---|-----------|------------|
| Description : | If the indicated bit is one, jump to the address indicated; otherwise proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the third instruction byte to the PC, after incrementing the PC to the first byte of the next instruction. The bit tested is not modified. No flags are affected. | | |
| Example : | <p>The data present at input port 1 is 11001010B. The Accumulator holds 56 (01010110B). The instruction sequence,</p> <pre>JB P1.2, LABEL1 JB ACC.2, LABEL2</pre> <p>will cause program execution to branch to the instruction at label LABEL 2.</p> | | |
| Operation : | $(PC) \leftarrow (PC) + 3$ IF (bit) = 1 THEN $(PC) \leftarrow (PC) + rel$ | | |

| JNB bit,rel | Function : Jump if Bit Not set | Bytes : 3 | Cycles : 2 |
|----------------------|--|-----------|------------|
| Description : | If the indicated bit is a zero, branch to the indicated address; otherwise proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the third instruction byte to the PC, after incrementing the PC to the first byte of the next instruction. The bit tested is not modified. No flags are affected. | | |
| Example : | <p>The data present at input port 1 is 11001010B. The Accumulator holds 56H (01010110B). The instruction sequence,</p> <pre>JNB P1.6, LABEL1 JNB ACC.3, LABEL2</pre> <p>will cause program execution to continue at the instruction at label LABEL2.</p> | | |
| Operation : | $(PC) \leftarrow (PC) + 3$ IF (bit) = 0 THEN $(PC) \leftarrow (PC) + rel$ | | |

JBC bit, rel Function : Jump if Bit is set and Clear bit Bytes : 3 Cycles : 2

Description : If the indicated bit is one, branch to the address indicated; otherwise proceed with the next instruction. The bit will not be cleared if it is already a zero. The branch destination is computed by adding the signed relative-displacement in the third instruction byte to the PC, after incrementing the PC to the first byte of the next instruction. No flags are affected.

Note : When this instruction is used to test an output pin, the value used as the original data will be read from the output data latch, not the input pin.

Example : The Accumulator holds 56 H (010010110B). The instruction sequence,

JBC ACC.3, LABEL1

JBC ACC.2, LABEL2

will cause program execution to continue at the instruction identified by the label LABEL2, with the Accumulator modified to 52 H (01010010B).

Operation :

$$(PC) \leftarrow (PC) + 3$$

IF (bit) = 1

THEN

$$(bit) \leftarrow 0$$

$$(PC) \leftarrow (PC) + rel$$

CJNE <dest-byte>,<src-byte>, rel Bytes : 3 Cycles : 2

Function : Compare and Jump if Not Equal

Description : CJNE compares the magnitudes of the first two operands, and branches if their values are not equal. The branch destination is computed by adding the signed relative-displacement in the last instruction byte to the PC, after incrementing the PC to the start of the next instruction. The carry flag is set if the unsigned integer value of <dest-byte > is less than the unsigned integer value of <src-byte>; otherwise, the carry is cleared. Neither operand is affected.

Example : The accumulator contains 45H. Register 7 contains 56H. The first instruction in the sequence.

```
CJNE R7, # 60H, NOT_EQ
```

```
;
```

```
; ..... ; R7 = 60H
```

```
NOT_EQ: JC REQ_LOW ; IF R7 < 60H
```

```
; ..... ; R7 > 60H.
```

Sets the carry flag and branches to the instruction at label NOT_EQ. By testing the carry flag, this instruction determines whether R7 is greater or less than 60H.

If the data being presented to Port 1 is also 45H, then the instruction.

```
WAIT : CJNE A, P1, WAIT
```

clears the carry flag and continues with the next instruction in sequence, since the Accumulator does equal the data read from P1. (If some other value was being input on P1, the program will loop at this point until the P1 data changes to 45H.)

CJNE A, direct, rel

Operation :

```
(PC) ← (PC) + 3
```

```
IF(A) <> (direct)
```

```
THEN
```

```
(PC) ← (PC) + relative offset
```

```
IF (A) < (direct)
```

```
THEN
```

```
(C) ← 1
```

```
ELSE
```

```
(C) ← 0
```

CJNE A, #data, rel

Operation :

```
(PC) ← (PC) + 3
```

```
IF(A) <> data
```

```
THEN
```

```
(PC) ← (PC) + relative offset
```

```
IF (A) < data
```

```
THEN
```

```
(C) ← 1
```


ELSE

 $(C) \leftarrow 0$ **CJNE Rn, #data, rel****Operation :** $(PC) \leftarrow (PC) + 3$ IF $(Rn) < > \text{data}$

THEN

 $(PC) \leftarrow (PC) + \text{relative offset}$ IF $(Rn) < \text{data}$

THEN

 $(C) \leftarrow 1$

ELSE

 $(C) \leftarrow 0$ **CJNE @R1, #data, rel****Operation :** $(PC) \leftarrow (PC) + 3$ IF $((R1)) < > \text{data}$

THEN

 $(PC) \leftarrow (PC) + \text{relative offset}$ IF $(R1) < \text{data}$

THEN

 $(C) \leftarrow 1$

ELSE

 $(C) \leftarrow 0$ **DJNZ <byte>, <rel-addr>****Bytes : 2/3****Cycles : 2****Function : Decrement and Jump if Not Zero****Description :**

DJNZ decrements the location indicated by 1, and branches to the address indicated by the second operand if the resulting value is not zero. An original value of 00H will underflow to 0FFH. No flags are affected. The branch destination would be computed by adding the signed relative-displacement value in the last instruction byte to the PC, after incrementing the PC to the first byte of the following instruction.

The location decremented may be a register or directly addressed byte.

Note : When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, not the input pins.

Example :

Internal RAM locations 40H, 50H, and 60H contain the values 01H, 70H, and 15H, respectively. The instruction sequence.

DJNZ 40H, LABEL_1

DJNZ 50H, LABEL_2

DJNZ 60H, LABEL_3

will cause a jump to the instruction at label LABEL_2 with the values 00H, 6FH and 15H in the three RAM location. The first jump was not taken because the result was zero.

The instruction provides a simple way of executing a program loop a given number of times, or for adding a moderate time delay (from 2 to 512 machine cycles) with a single instruction. The instruction sequence,

```

                                MOV          R2 #8
TOGGLE :                       CPL          P1.6
                                DJNZ         R2,TOGGLE

```

will toggle P1.6 eight times, causing four output pulses to appear at bit 6 of output Port 1. Each pulse will last three machine cycles; two for DJNZ and one to alter the pin.

DJNZ Rn, rel

Operation :

$(PC) \leftarrow (PC) + 2$

$(Rn) \leftarrow (Rn) - 1$

IF $(Rn) > 0$ or $(Rn) < 0$

THEN

$(PC) \leftarrow (PC) + rel$

DJNZ direct, rel**Operation :**

$(PC) \leftarrow (PC) + 2$

$(direct) \leftarrow (direct) - 1$

IF $(direct) > 0$ or $(Rn) < 0$

THEN

$(PC) \leftarrow (PC) + rel$

8.5.3 CALL and Subroutines

There are two subroutine-call instructions. LCALL (Long Call) and ACALL (Absolute Call). Each increments the PC to the first byte of the following instruction, then pushes it onto the stack (low byte first). Saving both bytes increment the stack pointer by two. The subroutine's starting address is encoded in the same ways as LJMP and AJMP. The generic form of the call operation is the mnemonic CALL, which 8051 will translate into LCALL or ACALL as appropriate.

The return instruction RET pops the high and low-order bytes of the program counter successively from the stack, decrementing the stack pointer by two. Program execution continues at the address previously pushed : the first byte of the instruction immediately following the call.

CALL and Return Instructions

| ACALL addr11 | Function : Absolute Call | Bytes : 2 Cycles : 2 |
|----------------------|--|----------------------|
| Description : | ACALL unconditionally calls a subroutine located at the indicated address. The instruction increments the PC twice to obtain the address of the following instruction, then pushes the 16-bit result onto the stack (low-order byte first) and increments the Stack Pointer twice. The destination address is obtained by successively concatenating the five high-order bits of the incremented PC, opcode bits 7-5, and the second byte of the instruction. The subroutine called must therefore start within the same 2K block of the program memory as the first byte of the instruction following ACALL. No flags are affected. | |
| Example : | Initially SP equals 08H. The label "SUBRTN" is at program memory location 0345 H. After executing the instruction, ACALL SUBRTN at location 0125H, SP will contain 0AH, internal RAM locations 09H and 0AH will contain 27H and 01H, respectively, and the PC will contain 0345H. | |
| Operation : | ACALL $(PC) \leftarrow (PC) + 2$ $(SP) \leftarrow (SP) + 1$ $(SP) \leftarrow (PC_{7-0})$ $(SP) \leftarrow (SP) + 1$ $((SP)) \leftarrow (PC_{15-8})$ $(PC_{10-0}) \leftarrow \text{Page address}$ | |

| LCALL addr16 | Function : Long call | Bytes : 3 | Cycles : 2 |
|--------------|----------------------|-----------|------------|
|--------------|----------------------|-----------|------------|

Description : LCALL calls a subroutine located at the indicated address. The instruction adds three to the program counter to generate the address of the next instruction and then pushes the 16-bit result onto the stack (low byte first), incrementing the Stack Pointer by two. The high-order and low-order bytes of the PC are then loaded, respectively, with the second and third bytes of the LCALL instruction. Program execution continues with the instruction at this address. The subroutine may therefore begin anywhere in the full 64 kbyte program memory address space. No flags are affected.

Example : Initially the Stack Pointer equals 07H. The label "SUBRTN" is assigned to program memory location 1234H. After executing the instruction,

LCALL SUBRTN

at location 0123, the Stack Pointer will contain 09H, internal RAM locations 08H and 09H will contain 26H and 01H, and the PC will contain 1234H.

Operation : LCALL

$$(PC) \leftarrow (PC) + 3$$

$$(SP) \leftarrow (SP) + 1$$

$$((SP)) \leftarrow (PC_{7-0})$$

$$(SP) \leftarrow (SP) + 1$$

$$((SP) \leftarrow (PC_{15-8})$$

$$(PC) \leftarrow addr_{15-0}$$

| RET | Function : Return from subroutine | Bytes : 1 | Cycles : 2 |
|-----|-----------------------------------|-----------|------------|
|-----|-----------------------------------|-----------|------------|

Description : RET pops the high-and low-order bytes of the PC successively from the stack, decrementing the Stack Pointer by two. Program execution continues at the resulting address, generally the instruction immediately following an ACALL or LCALL. No flags are affected.

Example : The stack Pointer originally contains the value 0BH. Internal RAM locations 0AH and 0BH contain the values 23H and 01H, respectively. The instruction,

RET

will leave the Stack Pointer equal to the value 09H. Program execution will continue at location 0123H.

Operation :

RET

$$(PC_{15-8}) \leftarrow ((SP))$$

$$(SP) \leftarrow (SP) - 1$$

$$(PC_{7-0}) \leftarrow ((SP))$$

$$(SP) \leftarrow (SP) - 1$$

| RET | Function : Return from interrupt | Bytes : 1 | Cycles : 2 |
|-----|----------------------------------|-----------|------------|
|-----|----------------------------------|-----------|------------|

Description :

RETI pops the high- and low-order bytes of the PC successively from the stack, and restores the interrupt logic to accept additional interrupts at the same priority level as the one just processed. The Stack Pointer is left decremented by two. No other registers are affected; the PSW is not automatically restored to its per-interrupt status. Program execution continues at the resulting address, which is generally the instruction immediately after the point at which the interrupt request was detected. If a lower - or same-level interrupt had been pending interrupt is processed.

Example :

The stack Pointer originally contains the value 0BH. An interrupt was detected during the instruction ending at location 0122H. Internal RAM locations 0AH and 0BH contain the values 23H and 01H, respectively. The instruction, RETI will leave the Stack Pointer equal to 09H and return program execution to location 0123H.

Operation :

RETI

$$(PC_{15-8}) \leftarrow ((SP))$$

$$(SP) \leftarrow (SP) - 1$$

$$(PC_{7-0}) \leftarrow ((SP))$$

$$(SP) \leftarrow (SP) - 1$$

| NOP | Function : No Operation | Bytes : 1 | Cycles : 1 |
|----------------------|--|-----------|------------|
| Description : | Execution continues at the following instruction. Other than the PC, no registers or flags are affected. | | |
| Example : | It is desired to produce a low-going output pulse on bit 7 of Port 2 lasting exactly 5 cycles. A simple SETB/CLR sequence would generate a one-cycle pulse, so four additional cycles must be inserted. This may be done (assuming no interrupts are enabled) with the instruction sequence, | | |
| | <pre> CLR P2.7 NOP NOP NOP NOP SETB P2.7 </pre> | | |
| Operation : | $(PC) \leftarrow (PC) + 1$ | | |

8.6 Programming Examples

Program 1 : Program to load accumulator A, DPH and DPL with 30H

```

MOV A, #30H      ; Loads 30H in A register
MOV DPH, A       ; (DPH) ← (A)
MOV DPL, A       ; (DPL) ← (A)

```

Program 2 : Copy byte in SCON to register R3.

Method 1 : Using direct address for SCON (98 H).

```
MOV R3, 98H      ; Copy SCON to R3
```

Method 2 : Using direct address for SCON (98H) and R2 (02H).

```
MOV 03H, 98H     ; Copy SCON to R3
```

Method 3 : Using indirect address for R3.

```
MOV R1, # 03H    ; Initialize pointer to R3
MOV @ R1, 98H    ; Copy SCON to R3

```

Method 4 : Using PUSH instruction.

```

MOV 81H, # 02H   ; Set the SP to address 02H in RAM
PUSH 98H         ; Push SCON (98H) to address (03H)

```


Program 3 : Put the number 90H in R2 and R3.

Method 1 : Use immediate addressing mode.

```
MOV R2, #90H
MOV R3, #90H
```

Method 2 : Use immediate and register addressing.

```
MOV R2, #90H
MOV R3, R2
```

Program 4 : Add two 8 bit numbers.

```
MOV A, #30H      ; (A) ← 30
ADD A, #50H      ; (A) ← (A) + 50H
```

Program 5 : Add two 16 bit numbers.

```
MOV DPTR, #2040H ; (DPTR) ← 2040H (16-bit number)
MOV A, #2BH      ; (A) ← 2BH (lower byte of second
                  ; 16-bit number)
MOV B, #20H      ; (B) ← 20H (Higher byte of second
                  ; 16-bit number)
ADD A, DPL       ; Add lower bytes
MOV DPL, A       ; Save result of lower byte addition
MOV A, B         ; Get higher byte of second number
                  ; in A
ADDC A, DPH      ; Add higher bytes with any carry
                  ; from lower byte addition
MOV DPH, A       ; Save result of higher byte addition
```

Program 6 : Find the 2's complement of a number in R0.

```
MOV A, R0        ; (A) ← (R0)
CPL A            ; 1s complement A
ADD A, #01       ; Add 1 to it to get 2s complement
```

Program 7 : Unpacked the packed BCD number stored in the accumulator and save the result in R0 and R1 such that (R0) ← LSB and (R1) ← MSB.

```
MOV B, A         ; Save the packed BCD number
ANL A, #0FH      ; Mask upper nibble of BCD number
MOV R0, A        ; Save the lower digit
MOV A, B         ; Get the packed BCD number
ANL A, #0F0H     ; Mask lower nibble of BCD number
SWAP A           ; Exchange the lower and upper
                  ; nibbles
MOV R1, A        ; Save the upper digit.
```

Program 8 : Subtract two 8-bit numbers and exchange digits.

```
MOV A, #9F       ; Get the first number in A
MOV R0, #40      ; Get the second number in R0
CLR C            ; Clear carry
SUBB A, R0       ; A ← A - (R0)
SWAP A           ; Exchange digits
```

Program 9 : Subtract the contents of R1 of Bank0 from the contents of R0 of Bank2.

```

MOV PSW,#10      ; Select Bank2
MOV A,R0          ; (A) ← (R0) from Bank2
MOV PSW,#00      ; Select Bank 0
CLR C             ; Clear carry
SUBB A,R1         ; A ← A-(R1) from Bank0

```

Program 10 : Division two 8-bit numbers.

```

MOV A,#90         ; Get the first number in A
MOV B,#20         ; Get the second number in B
DIV AB            ; A ÷ B, Remainder in B and Quotient
                  ; in A

```

Program 11 : Multiply two 8-bit numbers.

```

MOV A,#BF         ; Get the first number in A
MOV B,#79         ; Get the second number in B
MUL AB            ; A × B, Higher byte of result in B
                  ; and lower byte of result in A

```

Program 12 : Generate BCD up counter and send each count to port A.

```

MOV A,#00         ; Initialize counter
BACK: MOV P1,A     ; Send count to port A
      ADD A,#01    ; Increment counter
      DA A        ; Decimal adjust the counter
      AJMP BACK   ; Jump BACK

```

Program 13 : Find the maximum number from a given 8-bit ten numbers.

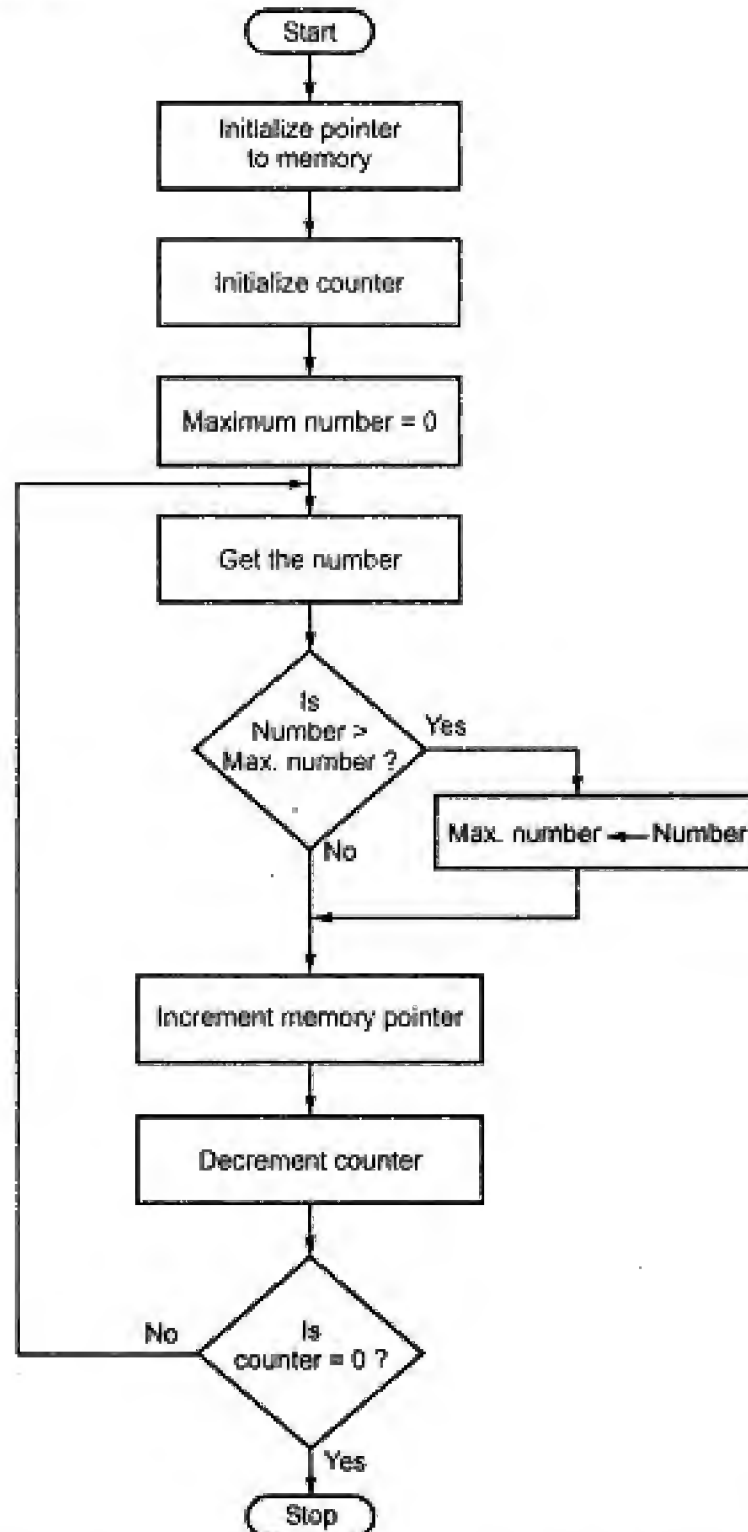
Flowchart : (See on next page)

Program :

```

MOV DPTR,#2000    ; Initialize pointer to memory where
                  ; numbers are stored
MOV R0,#0A        ; Initialize counter
MOV R3,#00        ; Maximum = 0
AGAIN: MOVX A,@DPTR ; Get the number from memory
      CJNE A,R3,NE  ; Compare number with maximum number
      AJMP SKIP    ; If equal go to SKIP
NE:    JC SKIP      ; If not equal check for carry,
                  ; if carry go to SKIP
      MOV R3,A      ; Otherwise maximum = number
SKIP:  INC DPTR     ; Increment memory pointer
      DJNZ R0,AGAIN ; Decrement count, if count = 0 stop
                  ; otherwise go to AGAIN

```



Program 14 : Program to convert 8-bit binary number to its equivalent BCD.

Program Logic :

Step 1 : Divide number with 100 decimal and save quotient i.e. save hundred's digit.

Step 2 : Make remainder as a new number.

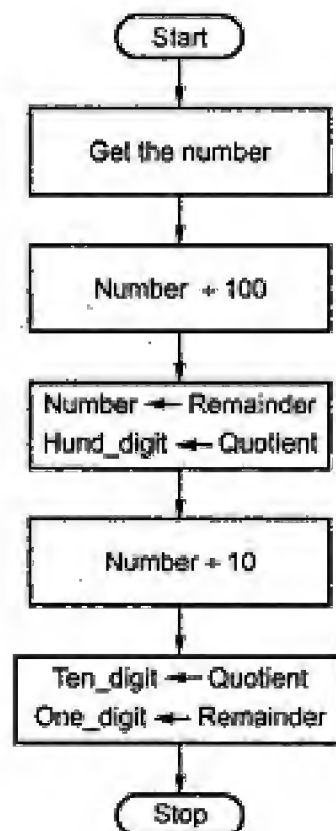
Step 3 : Divide number with 10 decimal and save quotient i.e. save tens digit.

Step 4 : Save remainder as ones digit.

Sample Example :

| | Quotient | Remainder |
|------------|----------|---------------------|
| 76 H + 100 | = 1 | 12 H |
| 12 H + 10 | = 1 | 8 |
| 76 H | = | (118) ₁₀ |

Flowchart :



Program :

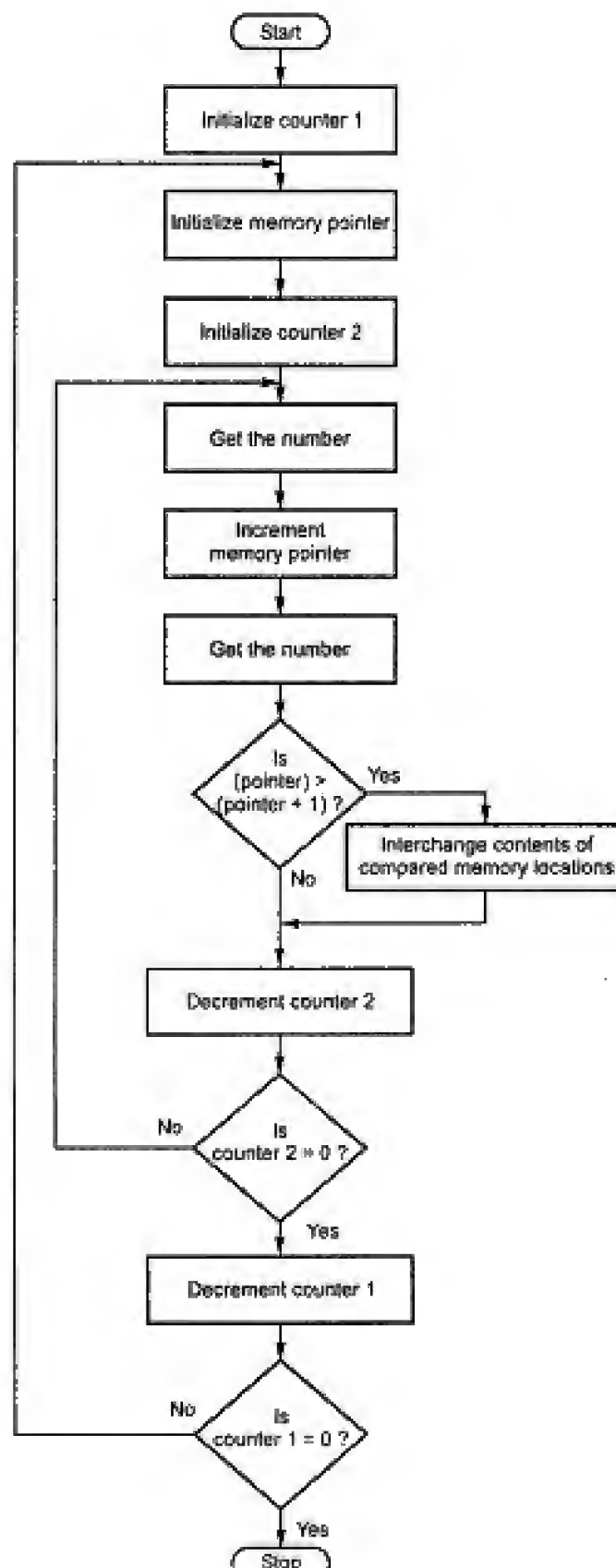
```

MOV A,#76H      ; Load the binary number in A
MOV B,#100      ; Load B with 100 decimal
DIV AB          ; Divide number with 100
MOV R0,A        ; Save the hundreds of the number
                ; (Quotient of the previous division)
MOV A,B         ; Get the remainder
MOV B,#10       ; Load B with 10 decimal
DIV AB          ; Divide number with 10

MOV R1,A        ; Save the tens of the number
MOV R3,B        ; Save the ones of the number
  
```

Program 15 : Arrange the given ten 8-bit numbers in the ascending order.

Flowchart :



Program :

```

      MOV R0,#09          ; Initialize counter1
AGAIN: MOV DPTR,#2000H    ; Initialize memory pointer
      MOV R1,#09          ; Initialize counter2
BACK:  MOV R2,DPL          ; Save lower byte of memory address
      MOVX A,@DPTR        ; Get the number
      MOV B,A             ; Save the number
      INC DPTR            ; Increment memory pointer
      MOVX A,@DPTR        ; Get the next number
      CJNE A,B,NE         ; If not equal check for greater or
                          ; less
      AJMP SKIP           ; Otherwise go to skip
NE:    JNC SKIP           ; If
      MOV DPL,R2          ; [ Exchange
      MOVX @DPTR,A        ; the contents
      INC DPTR            ; of two
      MOV A,B             ; memory
      MOVX @DPTR,A        ; locations ]
SKIP:  DJNZ R1,BACK       ; If R1 not equal to 0 go to BACK
      DJNZ R0,AGAIN       ; If R0 not equal to 0 go to AGAIN

```

Program 16 : Find the number of negative and positive numbers in a given array.

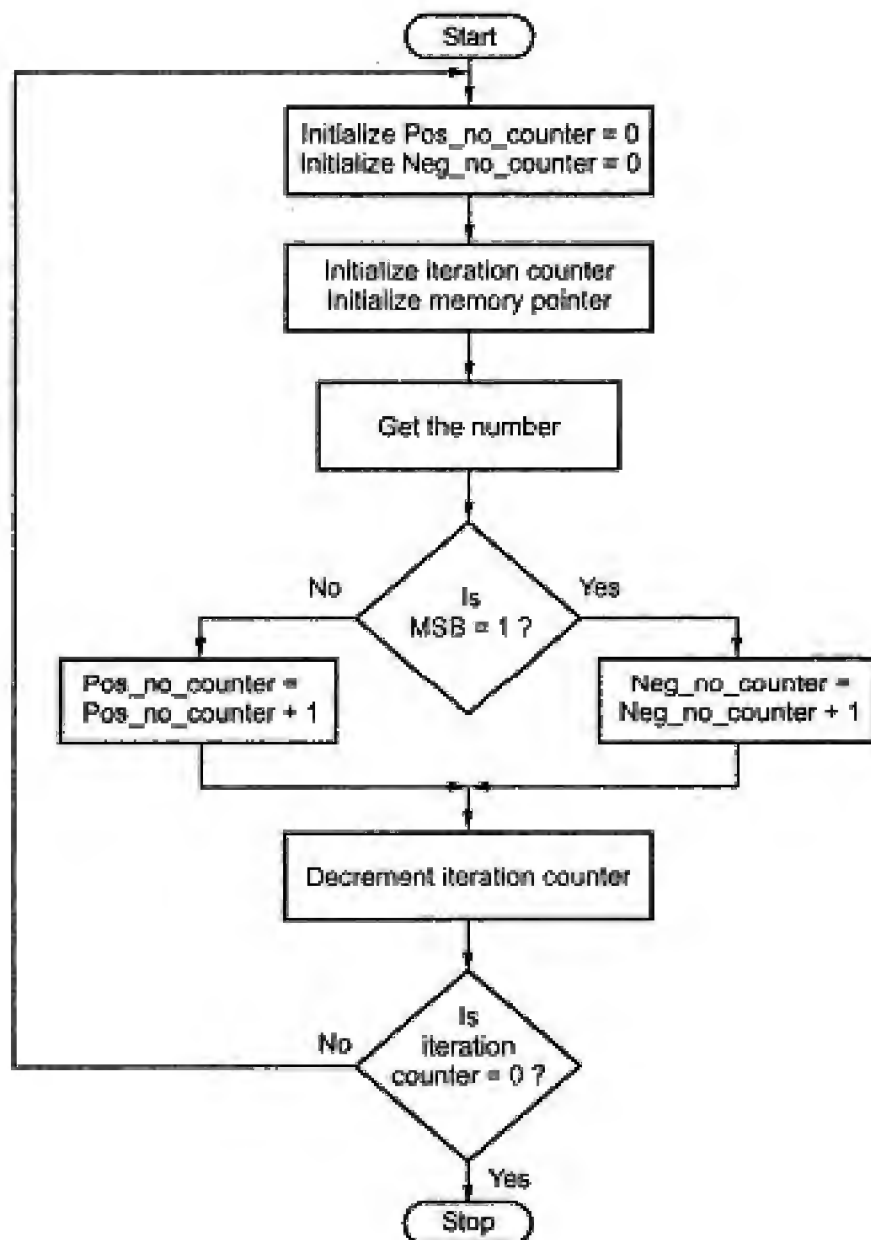
Flowchart : (See on next page)

Program :

```

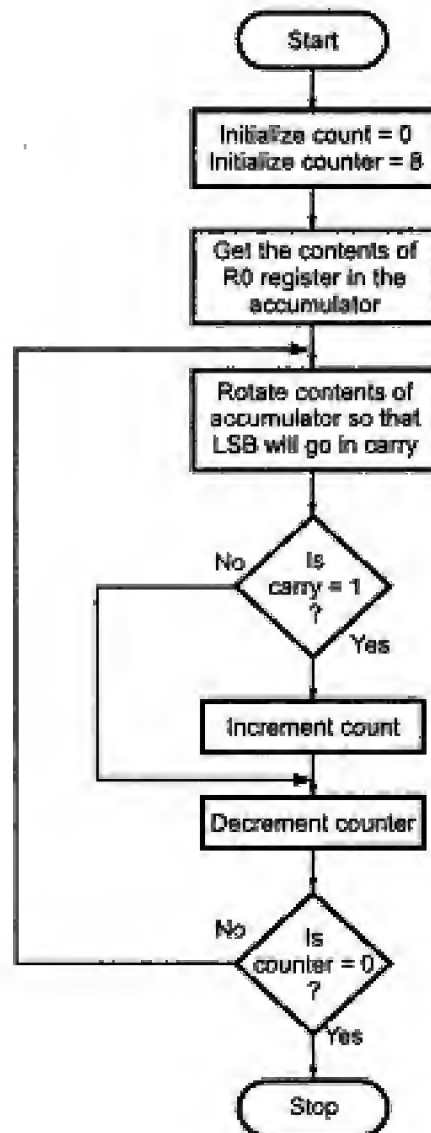
      MOV R0,#00          ; Initialize counter = 0 for
                          ; negative numbers
      MOV R1,R0           ; Initialize counter = 0 for
                          ; positive numbers
      MOV R2,#0AH         ; Initialize counter = 10
      MOV DPTR,#2000H     ; Initialize memory pointer
BACK:  MOVX A,@DPTR        ; Get the number
      ANL A,#80H          ; Mask lower 7 bits
      JZ SKIP             ; If Z = 0 goto SKIP
      INC R0              ; Otherwise increment negative number
                          ; count
      AJMP LAST           ; Go to LAST
SKIP:  INC R1              ; Increment positive number count
LAST:  DJNZ R2,BACK       ; Decrement R2 if not zero goto BACK

```

Program 17 : Count number of one's in a number.

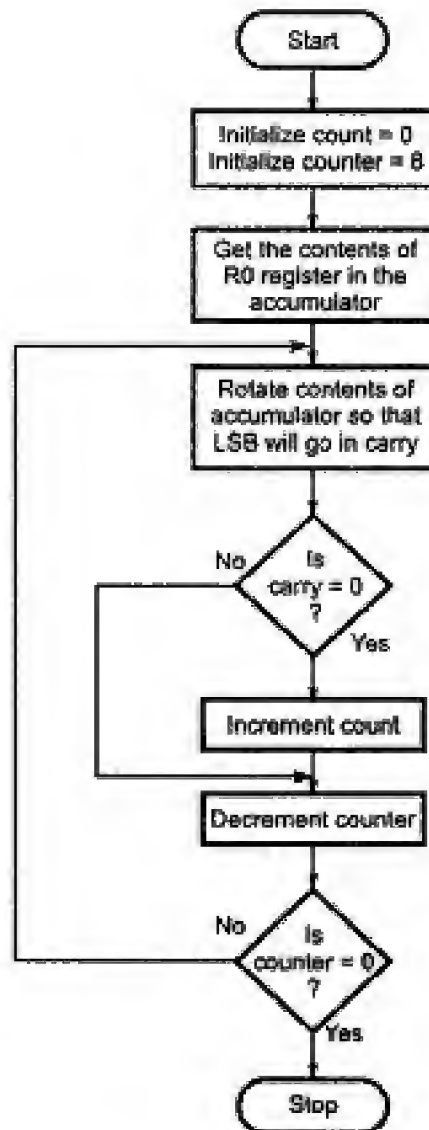
Flowchart :



| | | |
|------------------|---------------|--|
| Program : | MOV R2, #0 | ; Initialize one's counter = 0 |
| | MOV R1, #08 | ; Initialize iteration count |
| | MOV R0, #56 | ; Load number |
| | MOV A, R0 | ; Get the number in accumulator |
| BACK : | RRC A | ; Rotate A and CY ← LSB |
| | JNC SKIP | ; If carry is not zero go to skip |
| | INC R2 | ; Otherwise increment one's counter |
| SKIP : | DJNZ R1, BACK | ; Decrement iteration count and if not |
| | | ; zero repeat |

Program 18 : Count number of zero's in a number.

Flowchart :

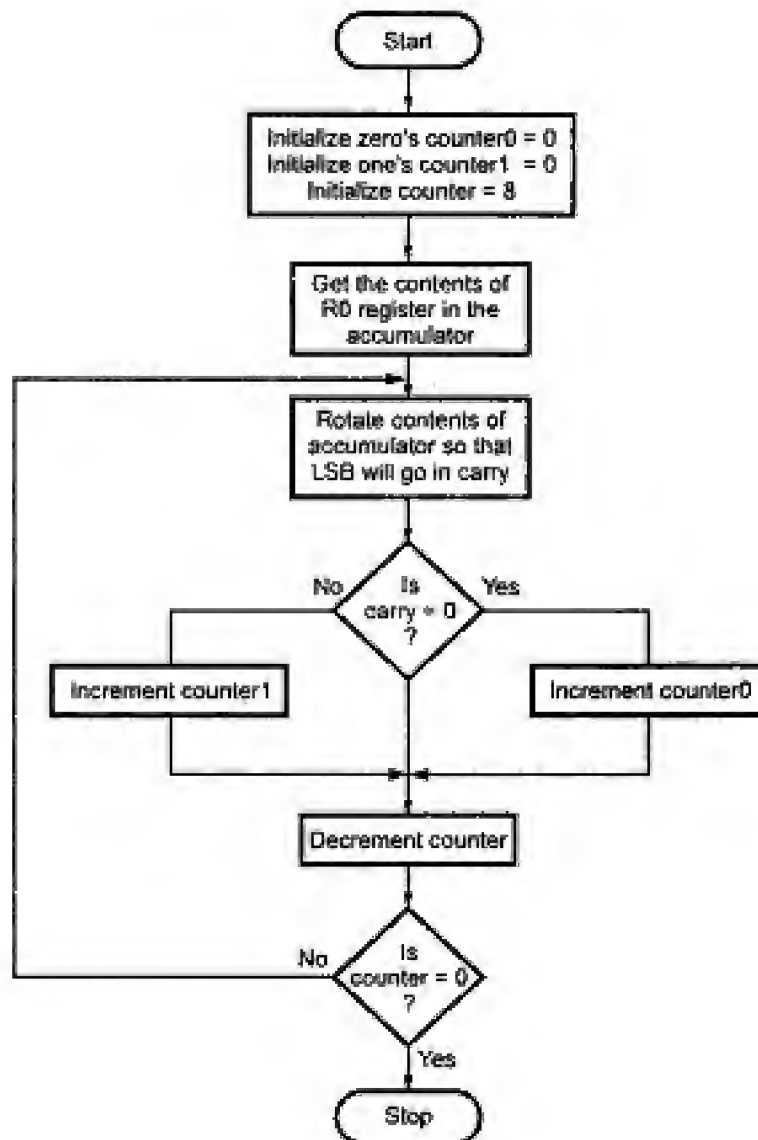


Program :

| | |
|----------------------|---|
| MOV R2, #0 | ; Initialize zero's counter = 0 |
| MOV R1, #08 | ; Initialize iteration count |
| MOV R0, #56 | ; Load number |
| MOV A, R0 | ; Get the number in accumulator |
| BACK : RRC A | ; Rotate A and CY ← LSB |
| JC SKIP | ; If carry is zero go to skip |
| INC R2 | ; Otherwise increment zero's counter |
| SKIP : DJNZ R1, BACK | ; Decrement iteration count and if not ; zero repeat |

Program 19 : Count number of one's and zero's in a number.

Flowchart :



Program :

| | | |
|--------|---------------|---|
| | MOV R2, #0 | ; Initialize one's counter = 0 |
| | MOV R3, #0 | ; Initialize zero's counter = 0 |
| | MOV R1, #08 | ; Initialize iteration count |
| | MOV R0, #56 | ; Load number |
| | MOV A, R0 | ; Get the number in accumulator |
| BACK : | RRC A | ; Rotate A and CY ← LSB |
| | JC SKIP | ; If carry is zero go to skip |
| | INC R3 | ; Otherwise increment zero's counter |
| | AJMP LAST | ; Go to last |
| SKIP : | INC R2 | ; Increment one's counter |
| LAST : | DJNZ R1, BACK | ; Decrement iteration count and if not zero repeat |

Program 20 : To generate a square wave on the port 1.

```

        MOV SP, #7H      ; Initialize stack pointer since we are
                          ; using subroutine program
BACK :  MOV P1, #00H      ; Send 00H on port 1 to generate low
                          ; level of square wave
        ACALL DELAY      ; Wait for sometime
        MOV P1, #0FFH    ; Send FFH on port 1 to generate high
                          ; level of square wave
        ACALL DELAY      ; Wait for sometime
        SJMP BACK        ; Repeat the sequence
DELAY : MOV R1, #0FFH    ; Load count
AGAIN : DJNZ R1, AGAIN   ; Decrement count and repeat the process
                          ; until Count is zero
        RET              ; Return to main program

```

Program 21 : To generate a square wave on the port pin P1.0.

```

        MOV SP, #7H      ; Initialize stack pointer since we are
                          ; using subroutine program
        CLR P1.0         ; Send 0 on port 1.0 to generate low
                          ; level of square wave
        ACALL DELAY      ; Wait for sometime
        SETB P1.0        ; Send 1 on port 1.0 to generate high
                          ; level of square wave
        ACALL DELAY      ; Wait for sometime
        SJMP BACK        ; Repeat the sequence
DELAY : MOV R1, #0FFH    ; Load count
AGAIN : DJNZ R1, AGAIN   ; Decrement count and repeat the process
                          ; until Count is zero
        RET              ; Return to main program

```

Program 22 : To find the sum of 10 numbers stored in the array.

Statement : Calculate the sum of series of numbers. The length of the series is in memory location 2200H and the series itself begins from memory location 2201H.

- Assume the sum to be 8 bit number so you can ignore carries. Store the sum at memory location 2300H.
- Assume the sum to be 16 bit number. Store the sum at memory locations 2300H and 2301H.

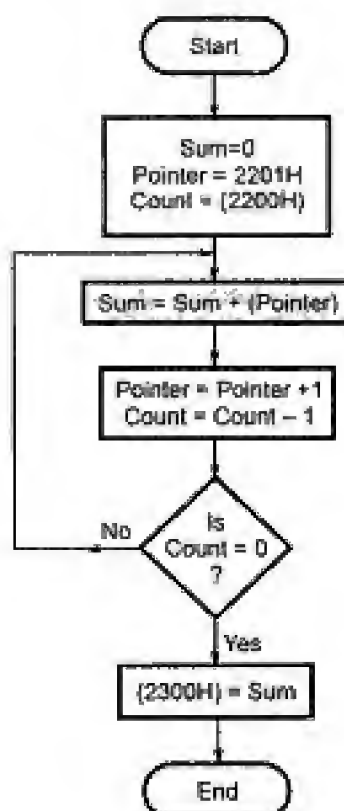
a. Sample problem

```

2200H    = 04H
2201H    = 20H
2202H    = 15H
2203H    = 13H
2204H    = 22H
Result   = 20 + 15 + 13 + 22 = 6AH
∴ 2300H  = 6AH

```

Flowchart :



Program :

```

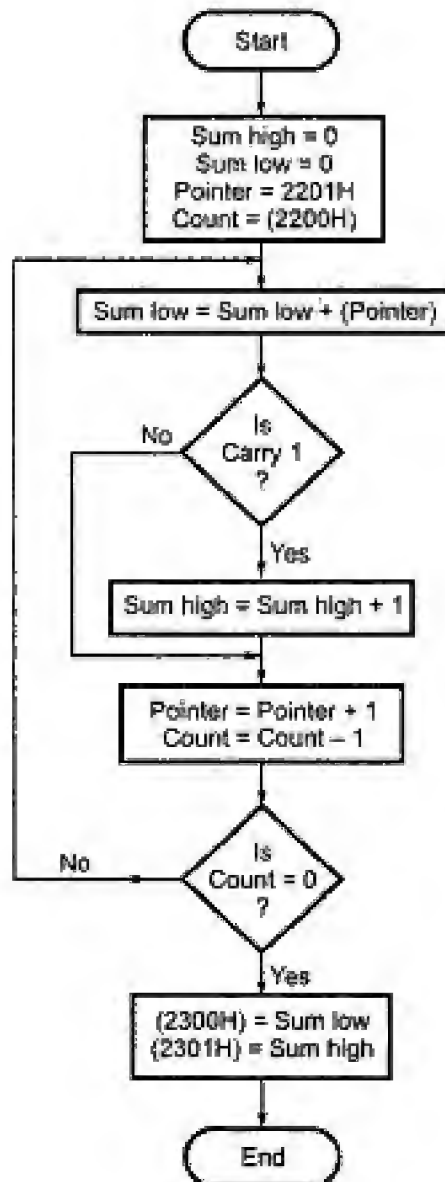
a)      MOV DPTR, #2200H ; Initialize memory pointer
        MOVX A, @DPTR   ; Get the count
        MOV R0, #10     ; Initialize the iteration counter
        INC DPTR        ; Initialize pointer to array of numbers
        MOV R1, #00     ; Result = 0
BACK :   MOVX A, @DPTR   ; Get the number
        ADD A, R1       ; A ← Result + A
        MOV R1, A       ; Result ← A
        INC DPTR        ; Increment the array pointer
        DJNZ R0, BACK   ; Decrement iteration count if not zero
        ; repeat
        MOV DPTR, #2300H ; Initialize memory pointer
        MOV A, R1       ; Get the result
        MOVX @DPTR, A   ; Store the result
  
```

b. Sample problem

```

2200H = 04H      2201H = 9AH
2202H = 52H      2203H = 89H      2204H = 3EH
Result = 9AH + 52H + 89H + 3EH = 1B3H
∴      2300H = B3H Lower byte  2301H = 01H Higher byte
  
```


Flowchart :



Program :

```

b)      MOV DPTR, #2200H ; Initialize memory pointer
        MOVX A, @DPTR   ; Get the count
        MOV R0, #10     ; Initialize the iteration counter
        INC DPTR        ; Initialize pointer to array of numbers
        MOV R2, #00     ; [Make
        MOV R1, #00     ; result = 00H]
BACK :   MOVX A, @DPTR   ; Get the number
        ADD A, R1       ; A ← Result + A
        MOV R1, A       ; Result ← A
        ADDC R2, #00    ; If carry exists, add it to MSD
        INC DPTR        ; increment the array pointer
        DJNZ R0, BACK   ; Decrement iteration count if not zero
                        ; repeat
  
```

```

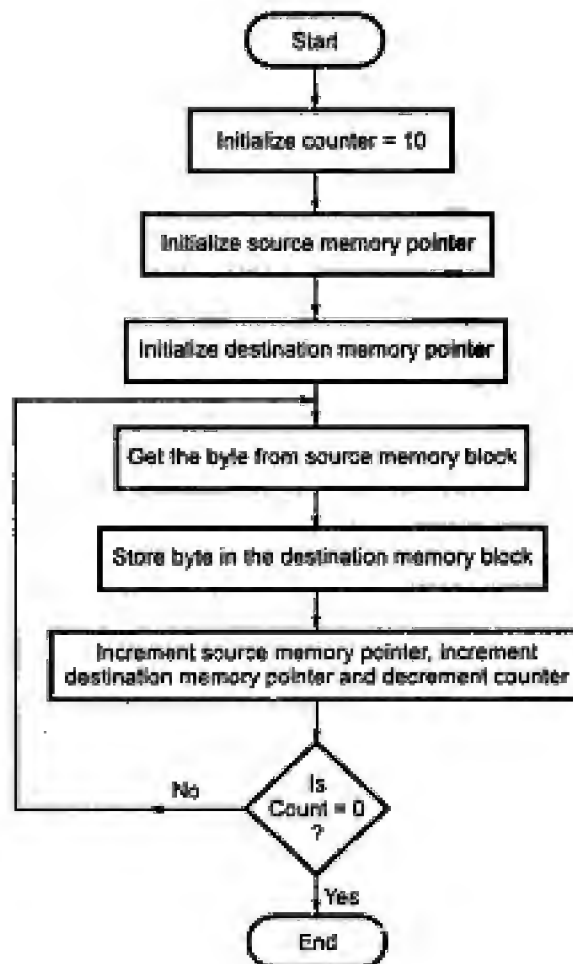
MOV DPTR, #2300H ; Initialize memory pointer
MOV A, R1         ; Get the lower byte of result
MOVX @DPTR, A     ; Store the lower byte of result
INC DPTR          ; Increment memory pointer
MOV A, R2         ; Get the higher byte of result
MOVX @DPTR, A     ; Store the higher byte of result

```

Program 23 : Data transfer from memory block B1 to memory block B2.

Statement : Assume two blocks are non-overlapped.

Flowchart :



Program :

```

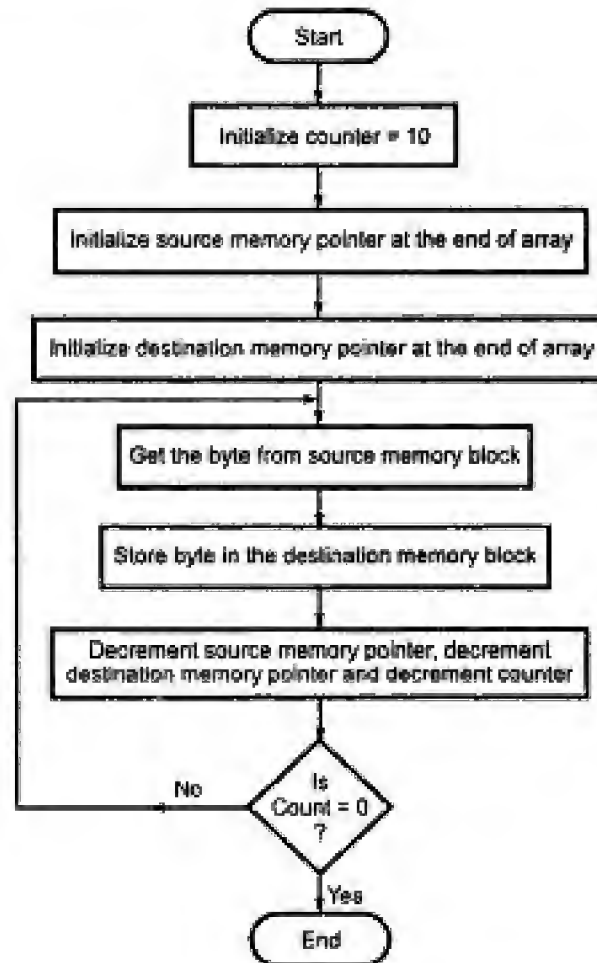
MOV R0, #10      ; Initialize iteration counter
MOV R1, #20H     ; Initialize source memory pointer
MOV R2, #30H     ; Initialize destination memory pointer
BACK : MOV A, @R1 ; Get data
      MOV @R2, A  ; Store data
      INC R1      ; Increment source memory pointer
      INC R2      ; Increment destination memory pointer
      DJNZ R0, BACK ; Decrement iteration count and if not
                  ; zero repeat

```

Program 24 : Data transfer from memory block B1 to memory block B2.

Statement : Assume two blocks are overlapped.

Flowchart :



Program :

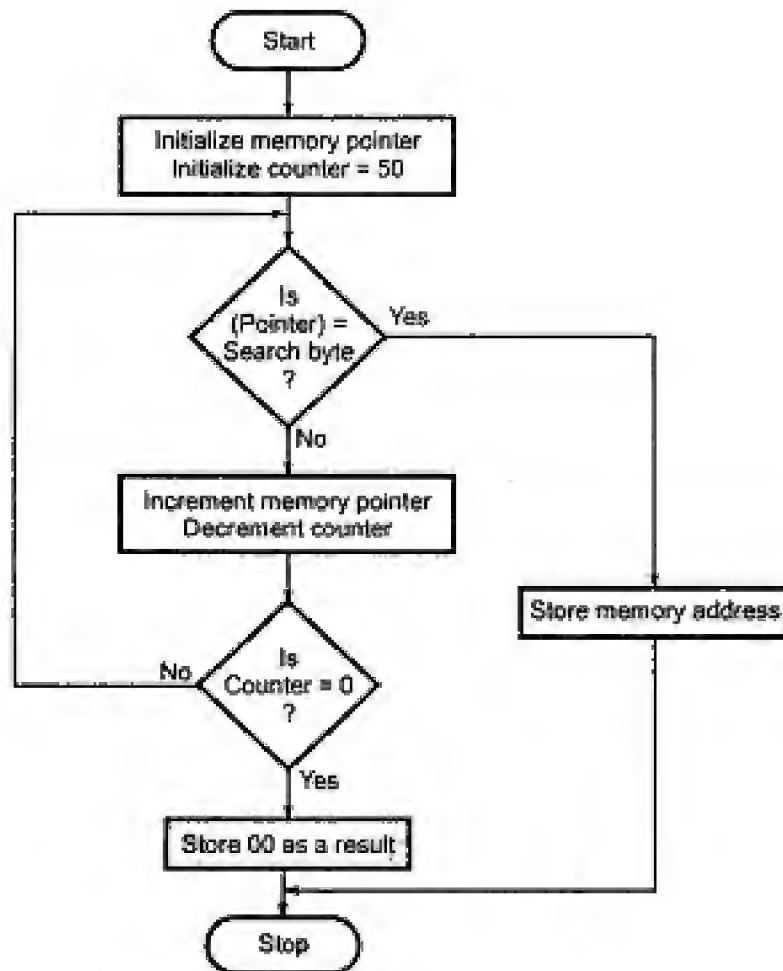
```

MOV R0, #10          ; Initialize iteration counter
MOV R1, #29H         ; Initialize source memory pointer
MOV R2, #32H         ; Initialize destination memory pointer
BACK : MOV A, @R1     ; Get data
      MOV @R2, A      ; Store data
      DEC R1          ; Decrement source memory pointer
      DEC R2          ; Decrement destination memory pointer
      DJNZ R0, BACK   ; Decrement iteration count and if not
                      ; zero repeat
  
```

Program 25 : To search a byte in a given numbers.

Statement : Search the given byte in the list of 50 numbers stored in the consecutive memory locations 2200H and 2201H. Assume that byte is 76H. If byte is not found store 00 at 2200 and 2201H.

Flowchart :



Program :

```

MOV R1, #00          ; [Load R1 and R2 with 00 so that if
MOV R2, #00          ; byte is not found we can load these
                     ; contents at 2200H and 2201H locations]
MOV R0, #50          ; Initialize iteration counter
MOV DPTR, #2000H     ; Initialize memory pointer
BACK : MOVX A, @DPTR  ; Get the number
      CJNE A, #76H, SKIP ; Search byte 76H if not equal go to SKIP
      MOV R1, DPL      ; [otherwise store
      MOV R2, DPH      ; the address of the byte]
      SJMP LAST
SKIP : INC DPTR
      DJNZ R0, BACK    ; Decrement iteration count and if not
                     ; zero repeat
LAST : MOV DPTR, #2200H ; Initialize memory pointer
      MOV A, R1        ; Get the lower byte of address
      MOVX @DPTR, A    ; Store the lower byte of address
      INC DPTR         ; Increment memory pointer
      MOV A, R2        ; Get the higher byte of address
      MOVX @DPTR, A    ; Store the higher byte of address

```

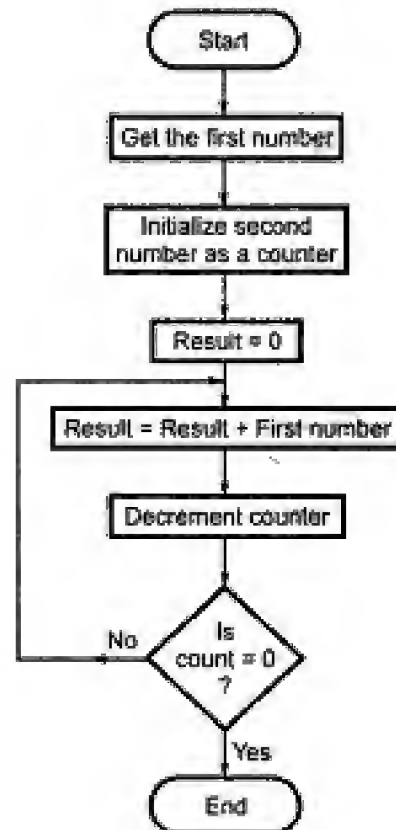
Program 26 : Multiply two 8-bit number using repetitive addition.

Statement : Multiply two 8-bit numbers stored in memory locations 2200H and 2201H. Store the result in memory locations 2300H and 2301H.

Sample problem

(2200H) = B2H
 (2201H) = 03H
 Result = B2H + B2H + B2H
 = 216H
 (2300H) = 16H
 (2301H) = 02H

Flowchart :



Program :

```

MOV DPTR, #2200H ; [Get the first
MOVX A, @DPTR    ; number]
MOV R0, A        ; Store the number
INC DPTR         ; [Get the second
MOVX A, @DPTR    ; number]
MOV R1, A        ; Store it as a counter
MOV R2, #00      ; [Make
MOV R3, #00      ; result = 0]
BACK : MOV A, R2  ; Get result (lower byte)
      ADD A, R0   ; Result = Result + number

```

```

MOV R2, A      ; Store result (lower byte)
ADDC R3, #00   ; If carry exists, add 1 to higher byte
DJNZ R1, BACK  ; Decrement counter, if not zero repeat
MOV A, R2      ; Get the lower byte of result
MOV DPTR, #2300H ; [Store the
MOVX @DPTR, A  ; Lower byte of result]
MOV A, R3      ; Get the higher byte of result
INC DPTR       ; [Store the
MOVX @DPTR, A  ; higher byte of result]

```

Program 27 : Binary to Gray Conversion

Statement : Write a program to convert a given 8-bit binary number into its Gray code equivalent

Let us see the Binary - Gray conversion process.

- Get the binary number.
- MSB of result \leftarrow MSB of the binary number.
- XOR bits from left to right with their adjacent bits.

Example :



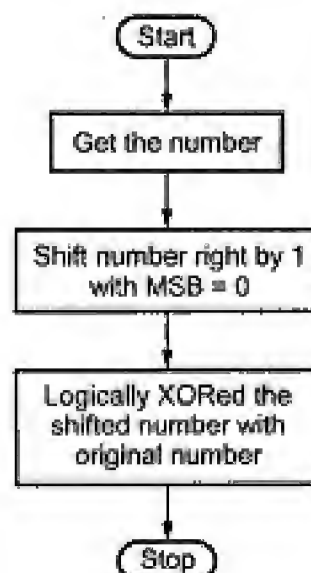
Program Logic :

- To XOR each bit with its adjacent bit we right shift the contents of original number and then XOR the result with the original number.

Example :

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|-----------------------------|
| | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | Binary number |
| ⊙ | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | Right shifted binary number |
| | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | Gray number |

Flowchart :



Program :

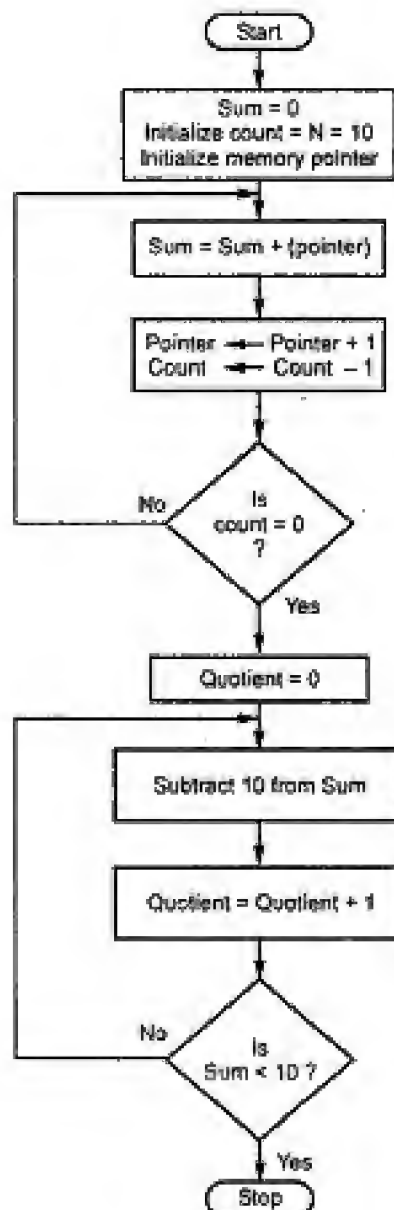
```

MOV A, #52H      ; Load binary number
MOV R0, A        ; Save binary number
CLR C            ; Clear carry flag so that after shifting
                  ; the contents we will get MSB = 0
RRC A            ; Right shift
XRL A, R0        ; XOR shifted contents with the original
                  ; number

```

Program 28 : To find the average of given N numbers**Program logic :**

- Calculate the sum of N numbers.
- Divide sum by the N.

Flowchart :

Program :

```

      MOV R0, #00      ; [Make
      MOV R1, #00      ; result = 0]
      MOV R2, #10      ; Initialize count
      MOV R3, #20H     ; Initialize memory pointer
BACK : MOV A, @R3       ; Get the number
      ADD A, R0         ; Add lower byte
      MOV R0, A         ; Save result
      ADDC R1, #00      ; If carry exists add Carry to higher
                        ; byte
      INC R3           ; Increment memory pointer
      DJNZ R2, BACK    ; Decrement counter and if not zero
                        ; repeat
      MOV R2, #00      ; Make quotient = 0
AGAIN : CLR C           ; Clear carry to perform subtraction
                        ; without borrow
      MOV A, R0         ; Get the lower byte
      SUBB A, #10H      ; Sub 10H from it
      MOV R0, A         ; Store result
      SUBB R1, #00H     ; If borrow exist subtract 1 from MSD
      INC R2           ; Increment quotient
      CJNE R1, #00, AGAIN ; If MSB not equal zero repeat
      MOV A, R0         ; [Otherwise check LSB
      CLR C             ; whether it is
      SUBB A, #10H      ; less than 10H]
      JNC AGAIN         ; If not repeat

```

Note : Quotient in R2 and Remainder in R0.

Program 29 : To add two 16-bit BCD numbers.

Flowchart : See on next page.

Program :

```

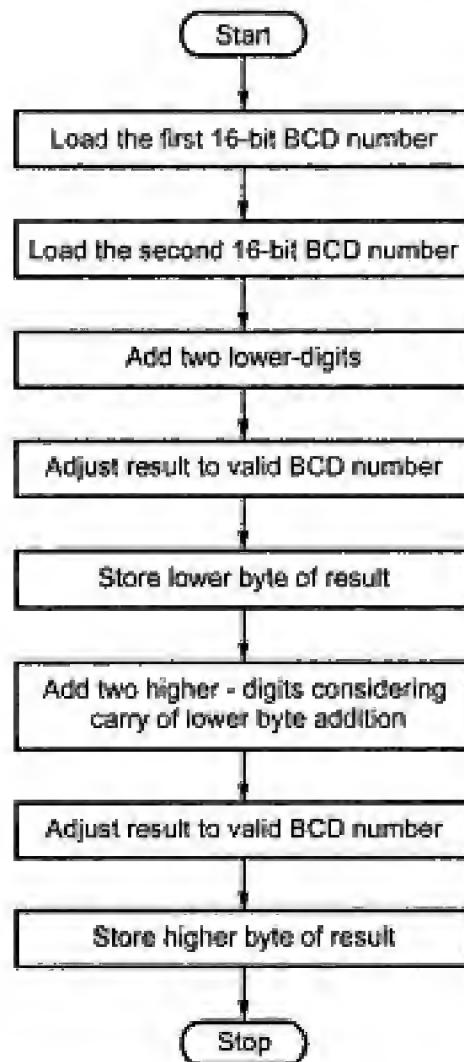
      MOV DPTR, #1234H ; Load first number
      MOV R0, #20H     ; Load lower byte of second number
      MOV R1, #30H     ; Load higher byte of second number
      MOV A, R0         ; Get the lower byte of second number
      ADD A, DPL         ; Add two lower bytes
      DA A              ; Adjust result to valid BCD
      MOV DPL, A        ; Store the sum of lower bytes

```

```

MOV A, R1          ; Get the higher byte of second
                   ; number
ADDC A, DPH         ; Add two higher bytes considering
                   ; carry of lower byte addition
DA A               ; Adjust result to valid BCD
MOV DPH, A         ; Store the sum of higher bytes

```



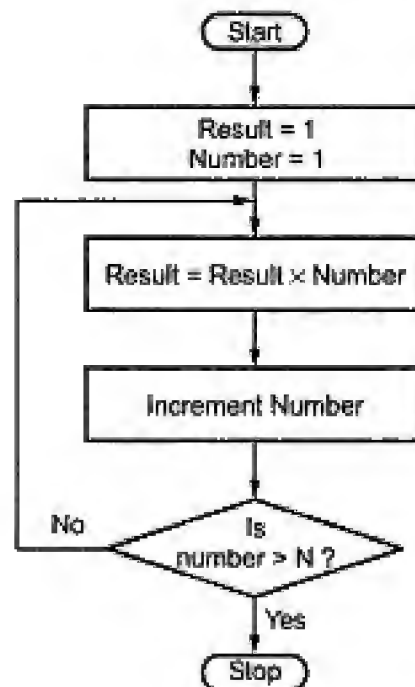
Program 30 : To find factorial of a number.

$$N! = N \times (N - 1) \times (N - 2) \times (N - 3) \times \dots \times 2 \times 1$$

For example :

$$\begin{aligned}
 6! &= 6 \times 5 \times 4 \times 3 \times 2 \times 1 \\
 &= 720
 \end{aligned}$$

Flowchart :



Program :

```

MOV A, #01      ; Store 01 in A register
MOV B, A        ; Store 01 in B register
CAL : MUL AB    ; Multiply two numbers
INC B           ; Increment number
CJNE B, #07H, CAL ; Check whether number is 7. If number is
                  ; 7 we have to stop

```

Note :

- This program can calculate factorial of numbers from 0 through 6.
- Result of factorial is available in A (LSB) and B (MSB) registers.

Program 31 : To find Fibonacci series of N given terms

The Fibonacci series is given as 0 1 1 2 3 5 8 13 21 ...

Hence we can write :

First term = 0

Second term = 1

Third term = First term + second term = 0 + 1 = 1

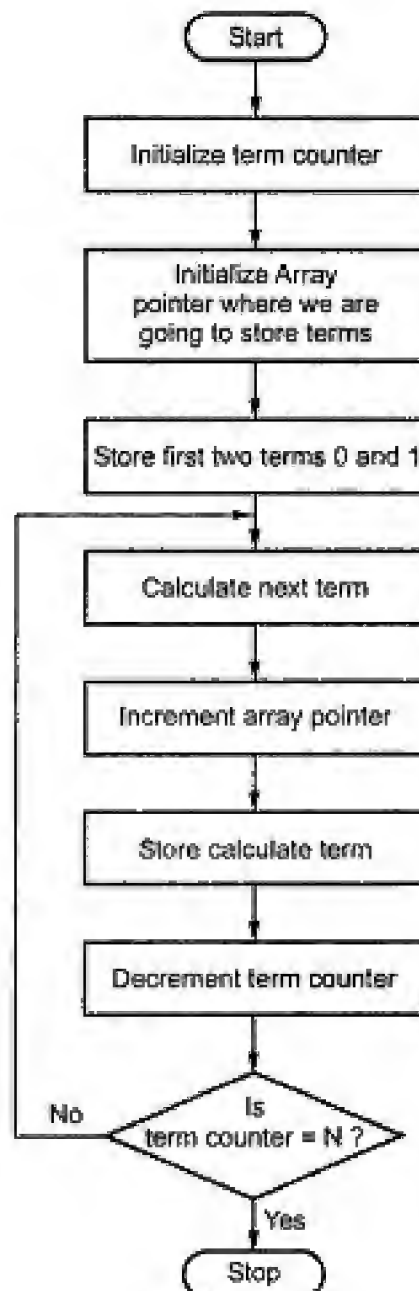
Fourth term = Second term + third term = 1 + 1 = 2

Fifth term = Third term + fourth term = 1 + 2 = 3

and so on. In general we can write

$$N^{\text{th}} \text{ term} = N - 2 \text{ term} + N - 1 \text{ term}$$

Flowchart :



Program :

| | |
|-------------------|--------------------------------------|
| MOV R0, #10 | ; Initialize term count |
| MOV R1, #20H | ; Initialize array pointer |
| MOV @R1, #0 | ; Store 1 st term as 0 |
| INC R1 | ; Increment memory pointer |
| MOV @R1, #1 | ; store 2 nd term as 1 |
| BACK : MOV A, @R1 | ; Get the current term |
| DEC R1 | ; Pointer to previous term |
| ADD A, @R1 | ; Add current term and previous term |

```

INC R1          ; {Point to
INC R1          ; next term}
MOV @R1, A      ; Store next term
CJNE @R1, 10, BACK ; Check whether last term calculated if
                  ; not repeat the process

```

Program 32 : Write an assembly language program to move 5 bytes of data stored at location 8000H onwards to the location C000H onwards and vice-versa

Program :

```

MOV R0, #05      ; Initialize count = 5
MOV DPL, #00H    ; Initialize memory pointer
MOV DPH, #80H    ; Adjust the memory pointer to
                  ; address the first memory
BACK: MOVX A, @DPTR ; Get data 1
      MOV R1, A     ; Save data 1 temporarily
      MOV DPH, #C0H ; Adjust the memory pointer to
                  ; address the second memory
      MOVX A, @DPTR ; Get data 2
      MOV R2, A     ; Save data 2 temporarily
      MOV A, R1     ; Get data 1
      MOVX @DPTR, A ; Save data 1
      MOV DPH, #80H ; Adjust memory pointer to
                  ; address the first memory
      MOV A, R2     ; Get data 2
      MOVX @DPTR, A ; Save data 2
      INC DPTR      ; Increment memory pointer
      DJNZ R0, BACK ; If count ≠ 0, repeat

```

Program 33 : An array of 10 numbers is stored at location 4000H onwards. Write an assembly language program to sort the numbers and store even numbers from location 5000H onwards and odd numbers from location 6000H onwards.

Program :

```

MOV R0, #0A      ; Initialize counter
MOV R1, #00H     ; Initialize memory pointer
MOV R2, #40H
MOV R3, #00H     ; Initialize even memory pointer
MOV R4, #50H
MOV R5, #00H     ; Initialize odd memory pointer
MOV R6, #60H
BACK: MOV DPL, R1 ; Load memory address
      MOV DPH, R2
      MOVX A, @DPTR ; Get data
      JB ACC.0, NEXT ; Check LSB of accumulator
                  ; if 1 goto NEXT (odd)
      MOV DPL, R3   ; Load even memory address
      MOV DPH, R4
      MOVX @DPTR, A ; Store even data
      INC R3        ; Increment even memory pointer
      SJMP NEXT1

```



```

NEXT:   MOV DPL, R5      ; Load odd memory address
        MOV DPH, R6
        MOVX @DPTR, A    ; Store odd data
        INC R5           ; Increment odd memory pointer
NEXT:   INC R1           ; Increment memory pointer
        DJNZ R0, BACK    ; If count ≠ 0, repeat

```

Program 34 : Write an assembly language program to realize following logic circuit using Boolean instructions of 8051.



Program : Let us assume that input a is connected to P1.0, input b is connected by P1.1 and output Y is connected at P2.0.

```

        MOV P1, #0FFH    ; Configure port1 as input
        MOV A, P1         ; Read port 1
        JB P1.0, NEXT     ; If P1.0 is set go to NEXT
        JB P1.1, NEXT1    ; If P1.1 is set goto NEXT1
        CLR P2.0          ; If P1.0=P1.1=0 make P2.0=0
        SJMP LAST
NEXT1:   SETB P2.0         ; If P1.0 = 0, P1.1=1 make P2.0=1
        SJMP LAST
NEXT:   JB P1.1, NEXT2    ; If P1.0=1, P1.1=0 make P2.0=1
        SETB P2.0
        SJMP LAST
NEXT2:   CLR P2.0         ; If P1.0 = 1, P1.1 = 1
                          ; make P2.0 = 0
LAST:   RET

```

Review Questions

1. Explain the addressing modes of 8051 with the help of examples.
2. Identify the addressing mode used by each of the following instruction.
 - a. MOV A, R4
 - b. MOV A, # 30H
 - c. ADD A, 40H
 - d. SWAP A
 - e. MOVC A, @ A + DPTR
3. Explain the instructions used to access external RAM.
4. Explain the instruction used to access external program memory.
5. Explain PUSH and POP instructions in 8051.
6. Explain the exchange instruction supported by 8051.
7. Explain byte level logical instructions of 8051.
8. Explain bit level logical instructions of 8051.
9. Explain rotate instructions of 8051.

10. *Explain addition and subtraction instructions of 8051.*
11. *Explain DAA instruction of 8051.*
12. *What is jump range ?*
13. *Explain various types of jump instructions according to range.*
14. *Explain CALL and RET instructions of 8051.*

□□□

Counter / Timer Programming in 8051

In the chapter 7, we have seen that 8051 has two Timers/Counters. They can be configured to operate either as timers or event counters. In this chapter we will study different modes of operation of these timers/counters and their programming.

9.1 8051 Timers

9.1.1 Timer Registers

Timer 0 and Timer 1 Registers

8051 has two timers, timer 0 and timer 1. Basically both, timer 0 and timer 1 are 16-bit registers. Since 8051 is an 8-bit microcontroller, each 16-bit register can be accessed as low-byte register (TL) and high-byte register (TH). Fig. 9.1 shows the timer 0 and timer 1 registers. These registers can be accessed like other registers (A, B, R0, R1 etc.) in 8051.

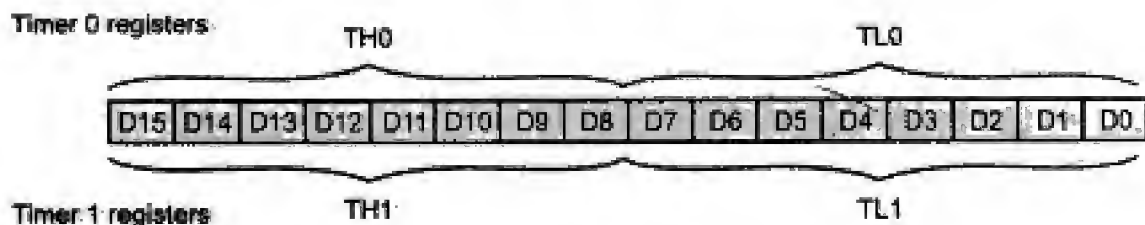


Fig. 9.1 Timer 0 and Timer 1 registers

Timer/Counter Mode Control (TMOD) Register

Timer/counter mode control (TMOD) is the special function register in 8051 having format as shown in Fig. 9.2.

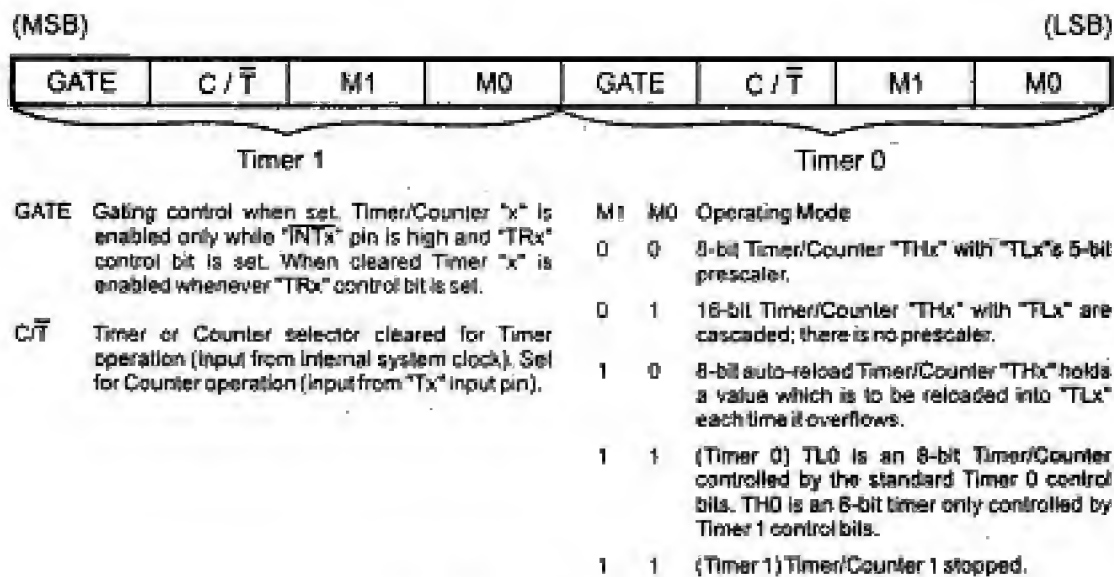


Fig. 9.2 TMOD register

Fig. 9.2 gives the function of all bits in TMOD register. Some more details of these bits are given below.

- **M1, M0** : These bits select the timer mode. There are four different modes of timer, mode 0, mode 1, mode 2 and mode 3. All these modes are discussed in the further section.

- **C / \bar{T}** : This bit is cleared ($C/\bar{T} = 0$) for selecting 'timer' operation and is set ($C/\bar{T} = 1$) for selecting 'counter' operation.

In the "Timer" mode, the register is incremented after every machine cycle. Thus, one can think of it as counting machine cycles. Since a machine cycle consists of 12 oscillator periods, the count rate is $\frac{1}{12}$ of the oscillator frequency. For example if crystal frequency is 12 MHz, then the timer clock frequency is 1 MHz.

In the "Counter" mode, the register is incremented in response to a 1-to-0 transition at its corresponding external input pin, T_0 or T_1 . In this mode, the external input is sampled during phase 2 of state 5 of every machine cycle. When the samples show a high in one cycle and a low in the next cycle, the count is incremented. The new count value appears in the register during phase 1 of state 3 of the cycle following the one in which the transition was detected. Since it takes 2 machine cycles (24 oscillator periods) to recognize a 1-to-0 transition, the maximum count rate is $\frac{1}{24}$ of the oscillator frequency. There are no restrictions on the duty cycle of the external input signal, but to ensure that a given level is sampled at least once before it changes, it should be held for at least one full machine cycle.

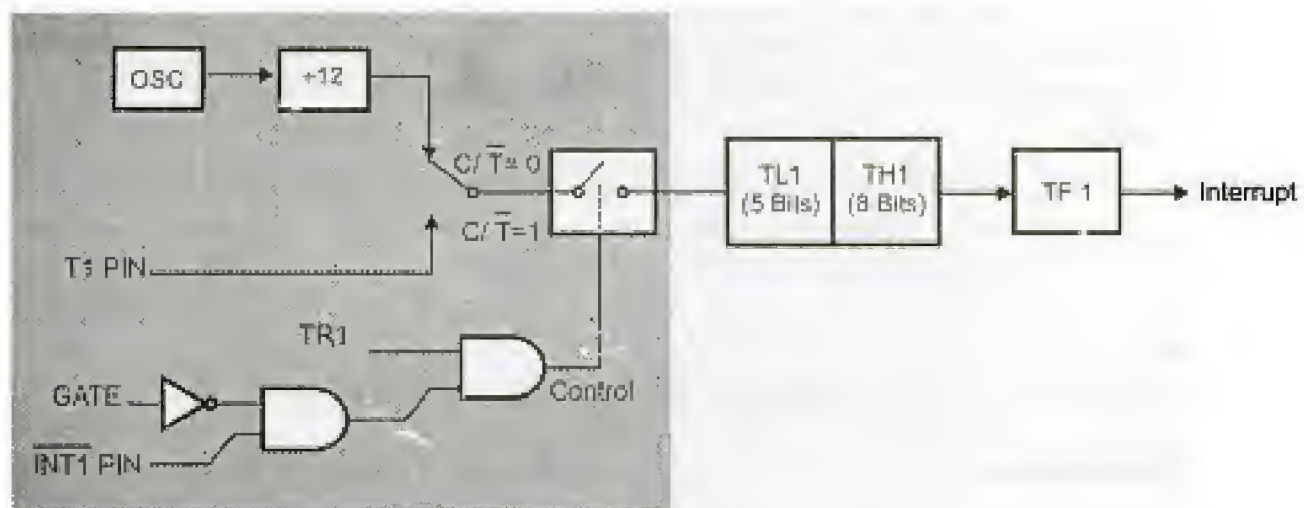
- **GATE** : The timers can be started and stopped by any one of the two methods, software or hardware. Sometimes, software as well as hardware both methods are available.

- **Software :** There are timer start bits (TR bits) in timer registers TR0 and TR1. The 'SETB TRx' instruction sets the TRx bit (TRx = 1) which starts the timer. x is 0 for timer 0 and is 1 for timer 1. The 'CLR TRx' instruction clears the TRx bit (TRx = 0) which stops the timer. Again, x is 0 for timer 0 and is 1 for timer 1. The timers can be started and stopped by 'SETB TRx' and 'CLR TRx' instructions respectively as long as 'GATE' bit in TMOD register is 0.
- **Hardware :** The timers can be started and stopped by an external hardware as long as 'GATE' bit in TMOD register is 1. In Figs. 9.5 and 9.6 , the start and stop operations of the timers are done externally through pins $\overline{\text{INT0}}$ (pin 3.2) and $\overline{\text{INT1}}$ (pin 3.3) for timers 0 and 1 respectively. The timer can be started or stopped at any time externally via a simple switch.

9.1.2 Programming 8051 Timers

As seen earlier, there are four modes of timer, mode 0, mode 1, mode 2 and mode 3. All these modes and their programming are discussed in this section. Mode 1 and mode 2 are widely used, so we will discuss them in detail.

9.1.2.1 Mode 0



Timer/ counter control logic

Fig. 9.3 Timer/counter 1 mode 0 : 13-bit counter

Both Timers in Mode 0 are 8-bit Counters with a divide-by-32 prescaler. This 13-bit timer is MCS-48 compatible. Fig. 9.3 shows the Mode 0 operation as it applies to Timer 1. In this mode, the Timer register is configured as a 13-bit register. As the count rolls over from all 1s to all 0s, it sets the Timer interrupt flag TF1. The counted input is enabled to the Timer when TR1 = 1 and either GATE = 0 or $\overline{\text{INT1}} = 1$. (Setting GATE = 1 allows the Timer to be controlled by external input $\overline{\text{INT1}}$, to facilitate pulse width measurements.) TR1 is a control bit in the Special Function Register TCON (Fig. 9.4) GATE is in TMOD.

| (MSB) | | | | (LSB) | | | |
|-------|-----|-----|-----|-------|-----|-----|-----|
| TF1 | TR1 | TF0 | TR0 | IE1 | IT1 | IE0 | IT0 |

| Symbol | Position | Name and Significance |
|--------|----------|--|
| TF1 | TCON.7 | Timer 1 Overflow Flag. Set by hardware on timer/counter overflow. Cleared when interrupt processed. |
| TR1 | TCON.6 | Timer 1 Run control bit. Set/cleared by software to turn timer/counter on/off. |
| TF0 | TCON.5 | Timer 0 Overflow Flag. Set by hardware on timer/counter overflow. Cleared when interrupt processed. |
| TR0 | TCON.4 | Timer 0 Run control bit. Set/cleared by software to turn timer/counter on/off. |
| IE1 | TCON.3 | Interrupt 1 Edge Flag. Set by hardware when external interrupt edge detected. Cleared when interrupt processed. |
| IT1 | TCON.2 | Interrupt 1 Type control bit. Set/cleared by software to specify falling edge/low level triggered external interrupts. |
| IE0 | TCON.1 | Interrupt 0 Edge Flag. Set by hardware when external interrupt edge detected. Cleared when interrupt processed. |
| IT0 | TCON.0 | Interrupt 0 Type control bit. Set/cleared by software to specify falling edge/low level triggered external interrupts. |

Fig. 9.4 TCON-timer/counter control/status register

The 13-bit register consists of all 8 bits of TH1 and the lower 5 bits of TL1. The upper 3 bits of TL1 are indeterminate and should be ignored. Setting the run flag (TR1) does not clear the registers.

Mode 0 operation is the same for Timer 0 as for Timer 1. Substitute TR0, TF0 and $\overline{\text{INT0}}$ for the corresponding Timer 1 signals in Fig. 9.3. There are two different GATE bits, one for Timer 1 (TMOD.7) and one for Timer 0 (TMOD.3).

9.1.2.2 Mode 1

Mode 1 is same as Mode 0, except that the timer register is being run with all 16 bits.

Mode 1 Programming

A time delay can be generated using mode 1 of the timer and following the steps given below.

1. Load TMOD register indicating which timer (timer 0 or timer 1) is used and which timer mode is selected.
2. Load TL and TH registers with count values.
3. Start the timer.
4. Monitor the timer flag (TF) with the 'JNB TFx, target address' instruction. When it is raised, get out of the loop.
5. Stop the timer.
6. Clear TF flag with 'CLR TFx' instruction.
7. Go back to step 2 for loading count value in TH and TL registers again.

9.1.2.3 Mode 2

Mode 2 configures the Timer register as an 8-bit Counter (TL1) with automatic reload, as shown in Fig. 9.5. Overflow from TL1 not only sets TF1, but also reloads TL1 with the contents of TH1, which is preset by software. The reload leaves TH1 unchanged. Mode 2 operation is the same for Timer/Counter 0.

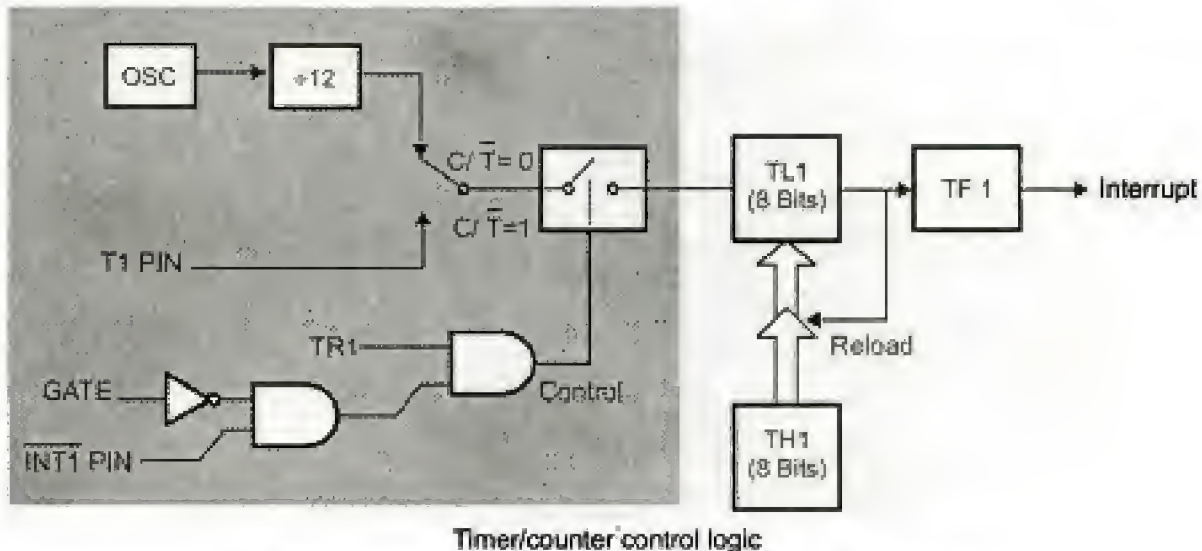


Fig. 9.5 Timer/counter 1 mode 2 : 8-bit auto reload

Mode 2 Programming :

A time delay can be generated using mode 2 of the timer and following are the steps given below.

1. Load TMOD register indicating which timer (timer 0 or timer 1) is used and which timer mode is selected.
2. Load TH register with count value.
3. Start the timer.
4. Monitor the timer flag (TF) with the 'JNB TFx, target address' instruction. When it is raised, get out of the loop.
5. Clear the TF flag, with 'CLR TFx' instruction.
6. Go back to step 4. There is no need to load TH register again since Mode 2 is auto reload.

9.1.2.4 Mode 3

Timer 1 in Mode 3 simply holds its count. The effect is the same as setting TR1 = 0. Timer 0 in Mode 3 establishes TL0 and TH0 as two separate counters. The logic for Mode 3 on Timer 0 is shown in Fig. 9.6. TL0 uses the Timer 0 control bits : C/ \bar{T} , GATE, TR0, $\overline{INT0}$, and TF0. TH0 is locked into a timer mode (counting machine cycles) and takes over the use of TR1 and TF1 from Timer 1. Thus, TH0 now controls the : Timer 1" interrupt.

Mode 3 is provided for applications requiring an extra 8-bit timer or counter. With Timer 0 in Mode 3, an 8051 can look like it has three Timer/Counters, and an 8052, like it has four. When timer 0 is in Mode 3, Timer 1 can be turned on and off by switching it out of and into its own Mode 3, or can still be used by the serial port as a baud rate generator, or in fact, in any application not requiring an interrupt.

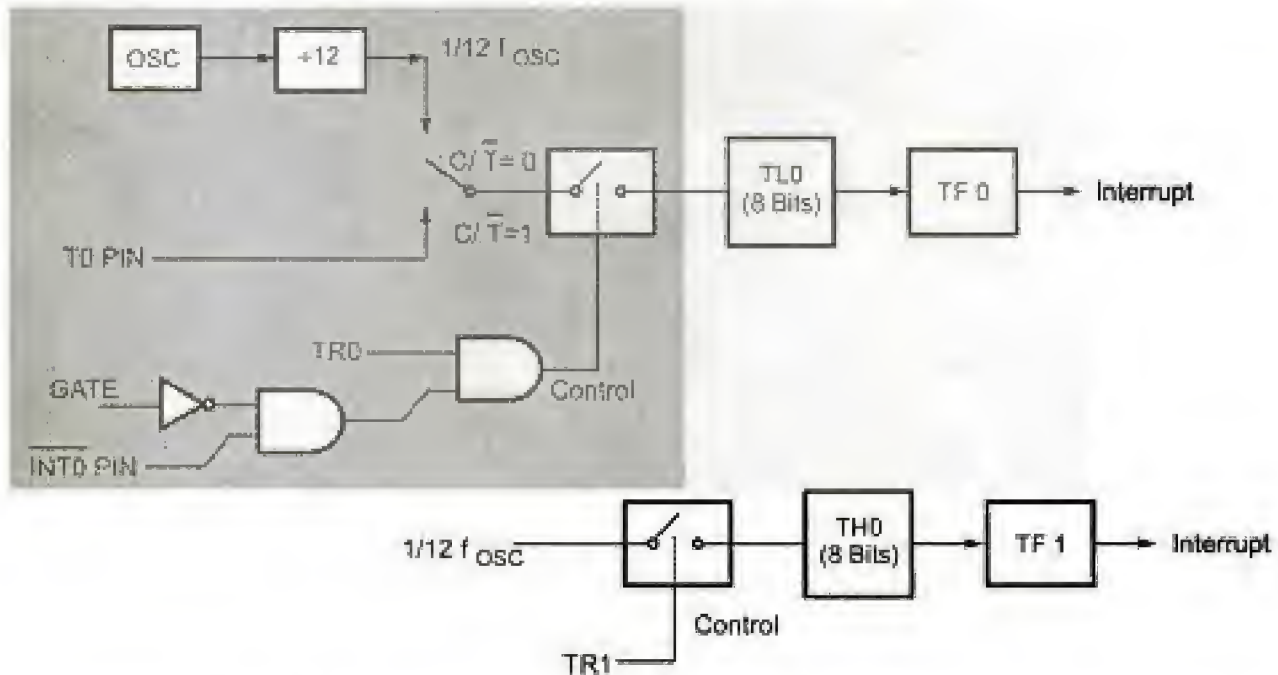


Fig. 9.6 Timer/counter 0 mode 3 : two 8-bit counters

The Table 9.1 summarizes the modes of timers.

| Mode | Brief Description |
|--------|--|
| Mode 0 | 13-bit timer (TL-5bits and TH-8bits). Counter overflow is indicated by time interrupt flag. |
| Mode 1 | 16-bit timer (TL-8bits and TH-8bits) Rest is same as mode 0. |
| Mode 2 | Automatic reload mode. 8-bit counter (TL-8bit) overflow from TL not only sets TF, but also reloads TL with the contents of TH. |
| Mode 3 | Establishes TL and TH as two separate counters. |

Table 9.1 Summary of timer modes

➡ **Example 9.1 :** State the mode in which timer will be operated after execution of following instructions.

i) `MOV TMOD, # 01H` ii) `MOV TMOD, # 12H`

Solution : From Fig. 9.7 we have,

- i) TMOD = 0000 0001, mode 1 of timer 0 will be selected.
- ii) TMOD = 0001 0010, mode 2 of timer 0, and mode 1 of timer 1 will be selected.

➡ **Example 9.2 :** Determine the timer's clock frequency and its period for 8051 systems with following crystal frequencies.

- i) 12 MHz ii) 11.0592 MHz

Solution :



Fig. 9.7

- i) $12 \text{ MHz} \div 12 = 1 \text{ MHz}$ and $T = 1/1\text{MHz} = 1 \mu\text{s}$
- ii) $11.0592 \div 12 = 921.6 \text{ kHz}$ and $T = 1/921.6 \text{ kHz} = 1.085 \mu\text{s}$

➡ **Example 9.3 :** If crystal used for 8031 microcontroller is of 4.5 MHz then determine the time for setting the timer flag for :

- i) Timer mode - 0 if THX starts of 00H
- ii) Timer mode - 1 if THX starts at 00H
- iii) Timer mode - 3 if count in THX is 9CH.

Solution : Timer clock's frequency = Crystal frequency \div 12

$$= 4.5 \text{ MHz} \div 12$$

$$= 375 \text{ kHz}$$

$$\text{Timer clock's period} = 1/375 \text{ kHz} = 2.667 \mu\text{s}$$

- i) In mode 0 timer is 13-bit hence the maximum count for timer is 1FFFH. When timer reaches its maximum of 1FFFH, it rolls over to 0000, and TF is raised.

\therefore Time for setting the timer flag is

$$= 2.667 \mu\text{s} \times 1\text{FFFH} (8191)$$

$$= 21.84 \text{ ms}$$

- ii) In mode 1 timer is 16-bit hence the maximum count for timer is FFFFH. When timer reaches its maximum of FFFFH, it rolls over to 0000, and TF is raised.

\therefore Time for setting the timer flag is

$$= 2.667 \mu\text{s} \times \text{FFFFH} (65535)$$

$$= 174.78 \text{ ms}$$

iii) In mode 3 timer 0 establishes TL0 and TH0 as two separate counters. Therefore, the maximum count for TH0 is FFH. When timer reaches its maximum of FFH, it rolls over to 00H, and TF1 is raised.

∴ Time for setting the TH1 is

$$= 2.667 \mu s \times (FFH - 9CH)$$

$$= 2.667 \mu s \times 63 H (99)$$

$$= 264 \mu s$$

9.1.3 Loading Timer Registers

Whenever, a time delay is to be generated using timer of 8051, it is the first step to find the counts which are to be loaded into the timer registers, TH and TL. These counts can be obtained by following the steps given below.

Let us assume the crystal frequency as F and desired time delay as t.

$$1. \text{ Timer period, } T = \frac{1}{F/12} = \frac{12}{F}$$

2. Obtain the decimal value, N by dividing the desired time delay(t) by time period(T).

$$\therefore N = \frac{t}{T}$$

3. Subtract N from 65536.

4. Convert result of subtraction (from step 4) to hex. Let this hex value be $C_3C_2C_1C_0$.

5. C_1C_0 is the count which is to be loaded into TL register and C_3C_2 in TH register.

$$\therefore TH = C_3C_2 \text{ and } TL = C_1C_0.$$

➡ **Example 9.4** : If the crystal frequency is 12 MHz, find the counts we need to load into the timer registers, if a required time delay is 5 ms. Also write a program to create a pulse having width of 5 ms on P1.5 using timer 0.

Solution : Crystal frequency = 12 MHz

$$\therefore T = \frac{12}{12 \times 10^6} = 1 \mu s.$$

i.e. the counter counts up every 1 μs . Out of many 1 μs intervals, we have to make a 5 ms pulse.

We need $5 \text{ ms} / 1 \mu s = 5000$ clocks.

$$N = 65536 - 5000 = 60536 = EC78H$$

We have to load TH with ECH and TL with 78H.

Program :

```

        CLR P1.5           ; Clear P1.5
        MOV TMOD, #01      ; Timer 0, mode 1(16-bit mode)
START : MOV TL0, #78H       ; TL0 = 78H, Timer 0 lower byte
                           ; register
        MOV TH0, #0ECH     ; TH0 = 0ECH, Timer 0 higher byte
                           ; register
        SETB P1.5          ; SET P1.5 high
        SETB TR0           ; start timer 0
REPEAT : JNB TF0, REPEAT   ; Monitor timer flag 0 till it becomes 1
        CLR TR0            ; Stop timer 0.
        CLR TF0            ; Clear timer 0 flag.

```

From all above discussion, it is clear that the time delay depends on the count loaded into the timer register and hence on the crystal frequency. The largest time-delay can be obtained by making this count 0(0000 H). If the time-delay which is to be produced is greater than this limit, then we can use one of the register of 8051 as a counter and a loop can be repeated as shown in the program given below. It creates multiple delays.

(The previous example with register R3, loaded with count 200 is taken).

```

        CLR P1.5
        MOV TMOD, #01
        MOV R3, #200       ; R3 used as a counter
                           ; to obtain multiple delays
START : MOV TL0, #10H
        MOV TH0, #03H
        SETB P1.5
        SETB TR0
REPEAT : JNB TF0, REPEAT
        CLR TR0
        CLR TF0
        DJNZ R3, START     ; If R3 is not zero,
                           ; reload timer register
                           ; and repeat the process.

```

Here,

Count in timer register is 0310 H = 784 in decimal.

$$65536 - 784 = 64752$$

$$64752 \times 1 \mu s = 64.752 \text{ ms}$$

For 200 repetitions, we get

$$200 \times 64.752 \text{ ms} = 12.95 \text{ seconds.}$$

9.2 Programming Counters

In the previous section we have seen, how the 8051's timer/counter can be operated as a timer to generate time delays. Also we have introduced the counter mode which can be selected by making C/T bit in TMOD register 1. In this mode the same timer is operated as an event counter.

Counter registers

When the timer/counter is used as a counter, the TMOD, TH and TL registers are used, functioning the same as for the timer studied in the last section.

C/T bit in TMOD register

As seen earlier, the C/ \bar{T} bit in the TMOD register decides the timer/counter functioning as a counter or a timer. When C/ \bar{T} bit in the TMOD register is 0, the timer mode is selected. When timer/counter is used as a timer, the 8051's crystal is used as a source of the frequency. When C/ \bar{T} bit in the TMOD register is 1, the counter mode is selected. When timer/counter is used as a counter, it gets its pulses from outside the 8051. The pin P3.4 (pin number 14) and pin 3.5 (pin number 15) of 8051 are used for applying pulses counter 0 and counter 1 respectively. These two pins belong to port 3. The counter counts up for each clock pulse applied at this pin. These pins are called T0 (timer 0 clock input) and T1(timer 1 clock input).

➡ **Example 9.5 :** Write a program for counter 1 in mode 2 to count the pulses and display the state of TL1 count on port 2. Assume that clock input is connected to T1 pin (P3.5).

Solution :

```

MOV TMOD, #01100000B ; Initialize counter 1 in
                        ; Mode 2, C/T=1
MOV TL1, #0           ; Clear TL1
SETB P3.5             ; Make T1 input
START : SETB TR1       ; Start the counter
BACK  : MOV A, TL1      ; Get the count from TL1
        MOV P2, A       ; Sent it to port 2
        JNB TF1, BACK    ; If TF1 = 0 repeat
        CLR TR1         ; Otherwise stop counter 1
        CLR TF1         ; Make TF1=0
        SJMP START      ; Repeat

```

Note : When 8051 is powered up ports are configured as input ports. To make them work as output port we have to send high output on it. Therefore, to behave T1 as input P3.5 is set.

➡ **Example 9.6 :** Write a program to display counter 0 on 7-segment LEDs. Assume that clock input is connected to pin (P 3.4)

Solution :

```

MOV TMOD, #00000110 ; Initialize counter 0 in
                        ; Mode 2, C/T=1
MOV TL0, #00H        ; Reset counter value
SETB P3.4             ; Make T0 as input
START : SETB TR0      ; Start counter 0
BACK  : MOV A, TL0     ; Get the count value
        ACALL CONVBCD
        MOV P2, A       ; Send count value in BCD on port 2
        JNB TF0, BACK    ; If TF0 = 0 repeat

```

```

        CLR TR0                ; Otherwise stop counter 0
        CLR TF0                ; Make TF0=0
        SJMP START             ; Repeat
CONVBBCD : ADD A, #00H         ; [Use DAA for
        DA A                   ; BCD conversion]
        RET                    ; Return to main program

```

The Fig. 9.8 shows the interface between 8051 and 7-segment displays.

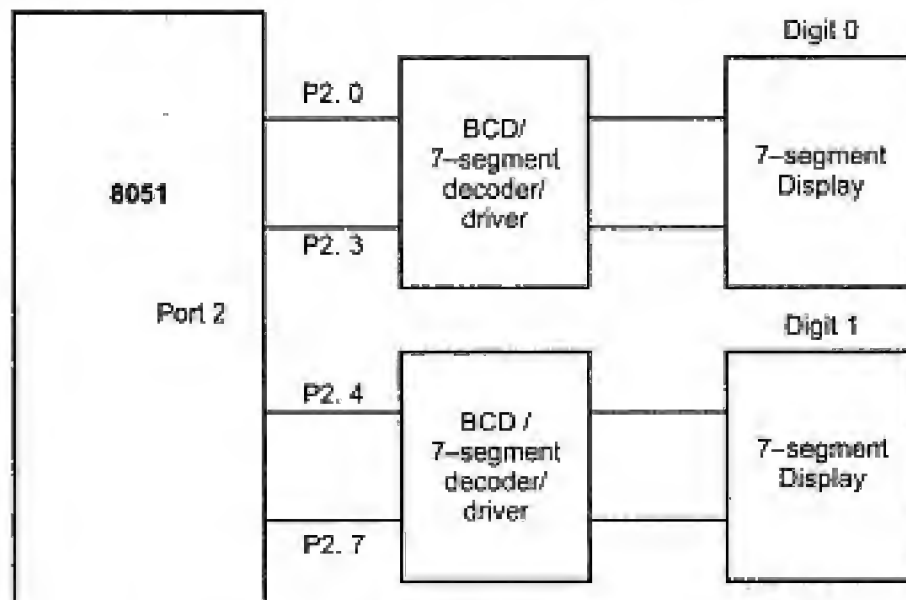


Fig. 9.8

➡ **Example 9.7 :** Using timer auto reload mode of 8051 generate a square wave of 2 kHz on port pin 1.0. Write an assembly language program for the same. Assume crystal frequency to be 11.0592 MHz.

Solution :

```

        MOV TMOD, #2H          ; Timer 0, mode 2
                                ; (8-bit, auto reload)
        MOV TH0, #19H          ; TH0=(255 - 230) = 25 = 19H
AGAIN:  ACALL DELAY             ; Wait for 0.25 ms
        CPL P1.0               ; Toggle P1.0
        SJMP AGAIN             ; Repeat
DELAY:  SET TR0                ; Start the timer 0
BACK:   JNB TF0, BACK          ; Stay until timer rolls over
        CLR TR0               ; Stop timer 0
        CLR TF0               ; Clear TF for next iteration
        RET

```

$$T_{ON} = T_{OFF} = (230 \times 1.085 \times 10^{-6}) \approx 2.5 \times 10^{-4}$$

$$\therefore T = 5 \times 10^{-4}$$

$$\therefore F = \frac{1}{T} = 2 \times 10^3 = 2 \text{ kHz}$$

➡ **Example 9.8 :** Write assembly language program for 8051 such that LED connected to port P1.0 flash at 0.5 sec rate when line P2.0 goes high. Use timer 0 for generating delay.

Solution : Program :

```

MOV TMOD, # 01      ; Timer 0, Mode 1 (16-bit mode)
MOV P2, #0FFH       ; configure P2 as input
CHECK: JB P2.0, CHECK ; Repeat until P2.0 = logic 1
HERE: CPL P1.0       ; Toggle P1.0
      ACALL DELAY     ; Wait for 0.5 sec
      SJMP HERE      ; Repeat

```

Assume XTAL = 12 MHz

∴ Timer clock frequency = $12 \text{ MHz} / 12 = 1 \text{ MHz}$

∴ $T = 1 \mu\text{s}$

With this timer frequency and 16-bit timer we can get maximum delay of $65536 \times 1 \mu\text{s} = 65.536 \text{ ms}$. Therefore, to get a delay of 0.5 sec. we have use external loop. We program timer 0 to give delay of 50 ms and such a delay is executed for 10 times to get a delay of 0.5 sec.

To get delay of 50 ms, the timer has to down step 50000 times. Therefore, the initial value to loaded in TH and TL will be

$$\begin{aligned} \text{Value} &= (65536 - 50000)_{10} \\ &= 3CB0\text{H} \end{aligned}$$

∴ $\text{TH} \leftarrow 3\text{CH}$ and $\text{TL} = \text{B0H}$

```

DELAY:  MOV R0, #0AH      ; Initialize counter to 10
BACK :  MOV TL0, #B0H     ; TL0 = B0H, the low byte
        MOV TH0, #3CH     ; TH0 = 3CH the high byte
        SETB TR0          ; Start the timer 0
AGAIN:  JNB TF0, AGAIN    ; Check timer0 flag until
        ; it rolls over
        CLR TR0           ; stop timer 0
        CLR TF0           ; Clear timer 0 flag
        DJNZ R0, BACK     ; Decrement counter and if not
        ; zero repeat
RET

```

Review Questions

1. Write a short note on timer registers in 8051.
2. Draw and explain the bit pattern of TMOD register.
3. Explain software and hardware methods to start and stop timers in 8051.
4. Explain, how to generate time delay using mode 1 of the timer.
5. Give steps to generate time delay using mode 2 of the timer.
6. Write a short note on counter registers in 8051.



8051 Serial Communication

Most of the microprocessors/microcontroller are designed for parallel communication. In parallel communication number of lines required to transfer data depend on the number of bits to be transferred. For example, to transfer a byte of data, 8 lines are required and all 8-bits are transferred simultaneously. Thus for transmitting data over a long distance, using parallel communication is impractical due to the increase in cost of cabling. Parallel communication is also not practical for devices such as cassette tapes or a CRT terminal. In such situations, serial communication is used. In serial communication one bit is transferred at a time over a single line.

Fig. 10.1 shows parallel and serial data transfers.

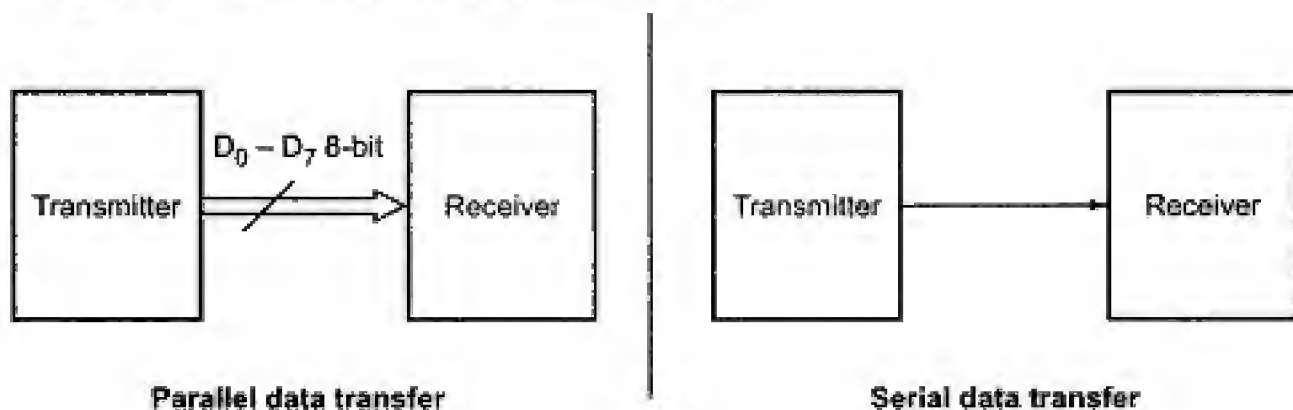


Fig. 10.1

In this chapter we will discuss the interfacing of 8051 to RS 232C and serial communication programming of 8051.

10.1 8051 Connections to RS 232C

In RS 232C the voltage level +3 V to +15 V is defined as logic 0; from -3 V to -15 V is defined as logic 1. The control and timing signals are compatible with the TTL level. Because of the incompatibility of the data lines with the TTL logic, voltage translators, called line drivers and line receivers, are required to interface TTL logic with the RS 232C signals. The Fig. 10.2 shows the connection between RS 232C and 8051. Here, MAX 232 chip is used as a line driver and line receiver. We know that 8051 assigns two pins RxD (P3.0) and TxD (P3.1) for reception and transmission of serial data, respectively. These pins

are TTL compatible; therefore, they require line driver and line receiver to make them RS 232C compatible. The MAX 232 has two sets of line drivers and line receivers for transmitting and receiving data. Only one set is required for one serial communication.

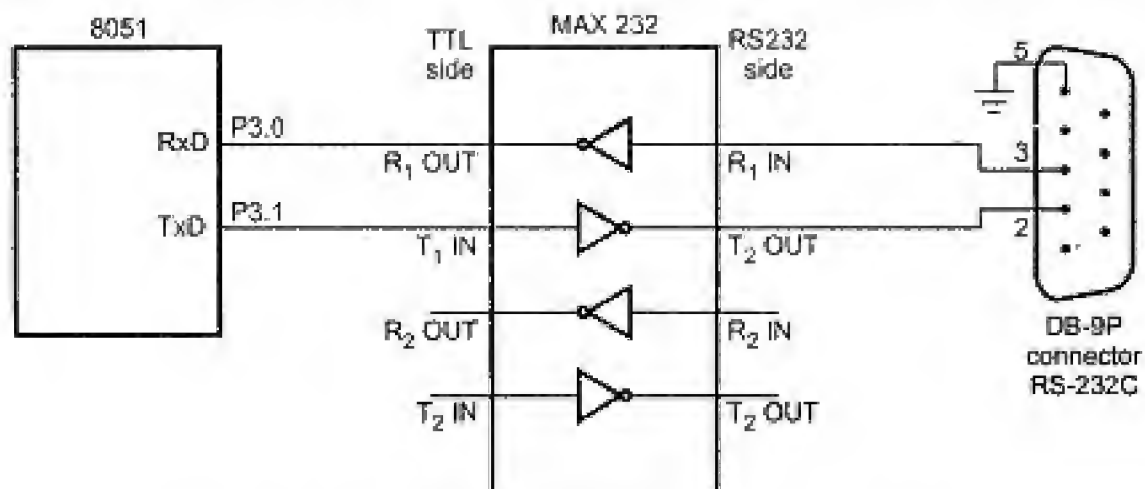


Fig. 10.2 Connection between RS-232C and 8051

10.2 8051 Serial Communication Programming

The serial port of 8051 is full duplex, means it can transmit and receive simultaneously. It can also receive-buffered, meaning it can commence reception of a second byte before a previously received byte has been read from the receive register. (However, if the first byte still hasn't been read by the time reception of the second byte is complete, one of the bytes will be lost). The serial port receives and transmits registers and both are accessed at Special Function Register SBUF. Writing to SBUF loads the transmit register, and reading SBUF accesses a physically separate receive register.

10.2.1 Operating Modes for Serial Port

MODE 0

In this mode, serial data enters and exits through RxD. TxD outputs the shift clock. 8-bits are transmitted/received : 8 data bits (LSB first). The baud rate is fixed at 1/12 the oscillator frequency.

MODE 1

In this mode, 10-bits are transmitted (through TxD) or received (through RxD) : a start bit (0), 8 data bits (LSB first), and a stop bit (1). On receiving, the stop bit goes into RB8 in Special Function Register SCON. The baud rate is variable.

MODE 2

In this mode, 11-bits are transmitted (through TxD) or received (through RxD) : a start bit (0), 8 data bits (LSB first), a programmable 9th data bit, and a stop bit (1). On Transmit, the 9th data bit (TB8 in SCON) can be assigned the value of 0 or 1. Or, for example, the parity bit (P, in the PSW) could be moved into TB8. On receive, the 9th data bit goes into RB8 in Special Function Register SCON, while the stop bit is ignored. The baud rate is programmable to either $\frac{1}{32}$ or $\frac{1}{64}$ the oscillator frequency.

MODE 3

In this mode, 11-bits are transmitted (through TxD) or received (through RxD) : a start bit (0), 8 data bits (LSB first), a programmable 9th data bit and a stop bit (1). In fact, Mode 3 is the same as Mode 2 in all respects except the baud rate. The baud rate in Mode 3 is variable.

In all four modes, transmission is initiated by any instruction that uses SBUF as a destination register. Reception is initiated in Mode 0 by the condition RI = 0 and REN = 1. Reception is initiated in the other modes by the incoming start bit if REN = 1.

The Table 10.1 summaries the four serial port modes provided by 8051.

| Mode | Transmission format | Baud rate |
|------|---|--|
| 0 | 8 data bits | $\frac{1}{12}$ oscillator frequency |
| 1 | 10-bit (start bit + 8 data bits + stop bit) | Variable |
| 2 | 11-bit (start bit + 8 data bits + programmable 9 th data bit + stop bit) | Programmable to either $\frac{1}{32}$ or $\frac{1}{64}$ oscillator frequency |
| 3 | 11-bit (start bit + 8 data bit + programmable 9 th data bit + stop bit) | Variable |

Table 10.1 Summary of serial port modes

10.2.2 SBUF

SBUF is an 8-bit register dedicated for serial communication in 8051. It can be accessed like any other register in 8051. The way in which SBUF is used for the transmission and reception of the data during serial communication is explained below.

- **Transmission** : When a byte of data is to be transmitted via the TxD pin, the SBUF is loaded with this data byte. As soon as a data byte is written into SBUF, it is framed with the start and stop bits and transmitted serially via the TxD pin.
- **Reception** : When 8051 receives data serially via RxD pin of it, the 8051 deframes it. The start and stop bits are separated out from a byte of data. This byte is placed in SBUF register.

10.2.3 Serial Port Control Register

The serial port control and status register is the Special Function Register SCON, shown in Fig. 10.3. This register contains not only the mode selection bits, but also the 9th data bit for transmit and receive (TB8 and RB8), and the serial port interrupt bits (TI and RI).

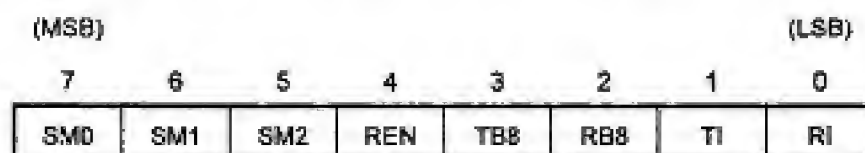


Fig. 10.3 Serial port control / status register

| Symbol | Position | Name and significance |
|--------|------------------------------|---|
| SM0 | SCON.7 | Serial port Mode control bit 0. Set/cleared by software (see note). |
| SM1 | SCON.6 | Serial port Mode control bit 1. Set/cleared by software (see note). |
| SM2 | SCON.5 | Serial port Mode control bit 2. Set by software to disable reception of frames for which bit 8 is zero. |
| REN | SCON.4 | Receiver Enable control bit. Set/cleared by software to enable/disable serial data reception. |
| TB8 | SCON.3 | Transmit Bit 8. Set/cleared by hardware to determine state of ninth data bit transmitted in 9-bit UART mode. |
| RB8 | SCON.2 | Receive Bit 8. Set/cleared by hardware to indicate state of ninth data bit received. |
| TI | SCON.1 | Transmit Interrupt flag. Set by hardware when byte transmitted. Cleared by software after servicing. |
| RI | SCON.0 | Receive Interrupt flag. Set by hardware when byte received. Cleared by software after servicing. |
| | Note : Mode | The state of (SM0, SM1) selects : |
| | 0 | 0 0 - Shift register ; baud = $f/12$ |
| | 1 | 0 1 - 8-bit UART, variable data rate. |
| | 2 | 1 0 - 9-bit UART, fixed data rate; baud = $f/32$ or $f/64$ |
| | 3 | 1 1 - 9-bit UART, variable data rate. |

Fig. 10.4 SCON–serial port control/status register

10.2.4 Generating Baud Rates

Serial Port in Mode 0 :

Mode 0 has a fixed baud rate which is 1/12 of the oscillator frequency. To run the serial port in this mode none of the Timer/Counters need to be set up. Only the SCON register needs to be defined.

$$\text{Baud rate} = \frac{\text{Osc freq}}{12}$$

Serial port in Mode 1

Mode 1 has a variable baud rate. The baud rate can be generated by either Timer 1 or Timer 2 (8052 only).

Using Timer/Counter 1 to Generate Baud Rates

For this purpose, Timer 1 is used in mode 2 (Auto-Reload).

$$\text{Baud rate} = \frac{k \times \text{Oscillator freq.}}{32 \times 12 \times [256 - \text{TH1}]}$$

If SMOD = 0, then k = 1.

If SMOD = 1, then k = 2. (SMOD is the PCON register)

Most of the time the user knows the baud rate and needs to know the reload value for TH1. Therefore, the equation to calculate TH1 can be written as :

$$\text{TH1} = 256 - \frac{k \times \text{Osc freq.}}{384 \times \text{Baud rate}}$$

TH1 must be an integer value. Rounding off TH1 to the nearest integer may not produce the desired baud rate. In this case, the user may have to choose another crystal frequency.

Since the PCON register is not bit addressable, one way to set the bit is logical ORing the PCON register. (i.e. ORL PCON, #80H). The address of PCON is 87H.

The Table 10.2 shows the values to be loaded into TH1 to get the corresponding baud rate. It also shows that the baud rates are doubled when SMOD = 1.

| TH1 (HEX) | Baud rate (SMOD = 0) | Baud rate (SMOD = 1) |
|---------------------------|----------------------|----------------------|
| FD | 9600 | 19,200 |
| FA | 4800 | 9600 |
| F4 | 2400 | 4800 |
| E8 | 1200 | 2400 |
| Note : XTAL = 11.0592 MHz | | |

Table 10.2

Using Timer/Counter 2 to Generate Baud Rates

For this purpose, Timer 2 must be used in the baud rate generating mode. If Timer 2 is being clocked through pin T2 (P1.0) the baud rate is :

$$\text{Baud rate} = \frac{\text{Timer 2 overflow rate}}{16}$$

And if it is being clocked internally the baud rate is :

$$\text{Baud rate} = \frac{\text{Osc freq.}}{32 \times [65536 - (\text{RCAP2H}, \text{RCAP2L})]}$$

To obtain the reload value for RCAP2H and RECAP2L the above equation can be rewritten as :

$$\text{RCAP2H}, \text{RCAP2L} = 65536 - \frac{\text{Osc freq.}}{32 \times \text{Baud rate}}$$

Serial Port in Mode 2

The baud rate is fixed in this mode and is $\frac{1}{32}$ or $\frac{1}{64}$ of the oscillator frequency depending on the value of the SMOD bit in the PCON register. In this mode none of the Timers are used and the clock comes from the internal phase 2 clock.

SMOD = 1, Baud rate = $\frac{1}{32}$ Osc freq.

SMOD = 0, Baud rate = $\frac{1}{64}$ Osc freq.

To set the SMOD bit : ORL PCON, #80H. The address of PCON is 87H.

Note : By changing SMOD bit in PCON from 0 to 1 we can double the baud rate in 8051.

Serial Port in Mode 3

The baud rate in mode 3 is variable and sets up exactly the same as in mode 1.

10.2.5 Programming 8051 for Serial Data Transfer

To program 8051, to transfer data serially we have to perform following sequence of actions :

1. Load the TMOD register with the value 20H to use timer 1 in mode 2 (8-bit auto-reload) to set the baud rate.
2. Load TH1 to set the desire baud rate for serial data transfer.
3. Load SCON register with the value 50H, to use serial mode 1, where an 8-bit data is framed with start and stop bits.
4. Set TR1 to 1 to start timer 1.
5. Clear TI with CLR TI instruction.
6. Write a character to be sent in to the SBUF register.

7. Check the TI flag bit with instruction `JNB TI, XXXX` to see if the character has been transferred completely.
8. Go to step 5 to transfer the next character.

➡ **Example 10.1 :** 8051 uses 11.0592 MHz crystal. To get 9600 hertz baud rate how will you program it for serial transmission ?

Solution : When 11.0592 MHz crystal is used and a standard baud rate of 9600 hertz is required then, the setting of TH_1 can be found as

$$\begin{aligned}
 TH_1 &= 256 - \frac{k \times \text{Osc freq}}{384 \times \text{Baud rate}} \\
 &= 256 - \frac{1 \times 11.0592 \times 10^6}{384 \times 9600} = 253 = FDH
 \end{aligned}$$

```

Program :   MOV TMOD, #020    ; Initialize timer 1 in mode 2
            MOV SCON, #4CH    ; Initialize serial mode 1
            ORL PCON, #80H    ; Make SMOD = 1
            MOV TH1, #FDH     ; Load count

```

➡ **Example 10.2 :** Write an 8051 assembly language program to transfer letter "G" serially at 9600 baud rate, continuously.

Solution :

```

            MOV TMOD, #20H    ; timer 1, mode 2 (auto reload)
            MOV TH1, #FDH    ; 9600 baud rate
            MOV SCON, #50H    ; 8-bit, 1 stop REN enabled
            SETB TR1         ; start timer 1
START:      MOV SBUF, # "G"   ; Letter "G" to be transferred
HERE:      JNB TI, HERE      ; Wait for the last bit to
                           ; transfer
            CLR TI           ; Clear TI for the next character
            SJMP START       ; Go to send the character again

```

➡ **Example 10.3 :** Write an 8051 assembly language program to transfer the message "HELLO" serially at 9600 baud, 8 bit data, 1 stop bit.

Solution :

```

            MOV TMOD, #20H    ; timer 1, mode 2
            MOV TH1, #FDH    ; 9600 baud rate
            MOV SCON, #50H    ; 8-bit, 1 stop bit, REN enabled
            SETB TR1         ; start timer 1
START:      MOV A, # "H"      ; transfer "H"
            ACALL TRANS
            MOV A, # "E"      ; transfer "E"
            ACALL TRANS
            MOV A, # "L"      ; transfer "L"
            ACALL TRANS
            MOV A, # "L"      ; transfer "L"
            ACALL TRANS

```



```

                MOV A, # "0"                ; transfer "0"
                ACALL TRANS                 ; Serial data transfer subroutine
TRANS:          MOV SBUF, A                ; Load SBUF
HERE:           JNB TI, HERE               ; wait for the last bit to
                                           ; transfer
                CLR TI                     ; Clear TI for the next
                                           ; character
                RET

```

10.2.5.1 Importance of the TI Flag Bit

When a data is to be transmitted via TxD pin, first a data byte is loaded into the SBUF register. A start bit, a data byte and then the stop bit are transmitted sequentially via TxD pin. During the transmission of the stop bit, 8051 sets the TI flag, i.e. TI = 1. This indicates the end of data byte transmission and 8051 is ready for the transmission and 8051 is ready for the transmission of next data. The programmer has to clear the TI flag, i.e. TI = 0, with the 'CLR TI' instruction to transmit next data. The TI flag bit should be monitored to make sure that the SBUF register is not overwritten. If we write the next byte to be transmitted into the SBUF register before setting the TI flag bit, the untransmitted portion of the previous byte will be lost. The programmer can check the TI flag bit by 'JNB TI, XX' instruction or by using an interrupt.

10.2.6 Programming 8051 for Receiving Serial Data

To program 8051, to receive data serially we have to perform following sequence of actions :

1. Load the TMOD register with the value 20H to use timer 1 in mode 2 (8-bit auto-reload) to set the baud rate.
2. Load TH1 to set the desire baud rate for serial data transfer.
3. Load SCON register with the value 50H, to use serial mode 1, where an 8-bit data is framed with start and stop bits.
4. Set TR1 to 1 to start timer 1.
5. Clear RI with CLR RI instruction.
6. Check the RI flag bit with instruction JNB RI, XXXX to see if an entire character has been received yet.
7. If RI is set, SBUF has the byte. Save this byte.
8. Go to step 5 to receive the next character.

➡ **Example 10.4 :** Write an 8051 assembly language program to receive bytes serially with baud rate 9600, 8-bit data and 1 stop bit. Simultaneously send received bytes to port 2.

Solution :

```

MOV TMOD, #20H    ; timer 1, mode 2 (auto reload)
MOV TH1,  #FDH    ; 9600 baud rate
MOV SCON, #50H    ; 8-bit, 1 stop, REN enabled

```

```
                SETB TR1                ; start timer 1
HERE:           JNB RI, HERE            ; wait for character receive
                ; completely
                MOV A, SBUF             ; save the received character
                MOV P2, A               ; send character to port 2
                CLR RI                 ; Get ready to receive next byte
                SJMP HERE               ; Go to receive next character
```

10.2.6.1 Importance of the RI Flag Bit

When a data is to be received via RxD pin, first the start bit, and a data byte with one bit at time are received sequentially via RxD pin. When the last bit of a data byte is received, a byte is formed and the SBUF register is loaded with this byte. Then the stop bit is received. During the reception of the stop bit, 8051 sets the RI flag, i.e. RI = 1. This indicates that the entire data byte has been received. This data byte which is loaded in the SBUF register, should be placed to a safe place such as any register or memory before it is lost. The programmer has to clear the RI flag, i.e. RI = 0, with the 'CLR RI' instruction to receive next data and place it in SBUF register. The RI flag bit should be monitored to make sure that the entire byte has been received. This monitoring of the RI flag bit till it becomes one to ensure the reception of complete data place it in SBUF register, copy it to the safe place and then make RI zero with 'CLR RI' instruction are the necessary steps to avoid any loss of received data.

10.2.7 Doubling the Baud Rate in the 8051

We can double the baud rate in 8051 using two way

- By doubling the crystal frequency.
- By making SMOD bit in the PCON register from 0 to 1.

➡ **Example 10.5 :** Write a program to receive message from PC to 8051. Message string is "Hello". After this microcontroller sends message to PC "Fine".

Solution : The Fig. 10.5 shows the connections between 8051 and PC.

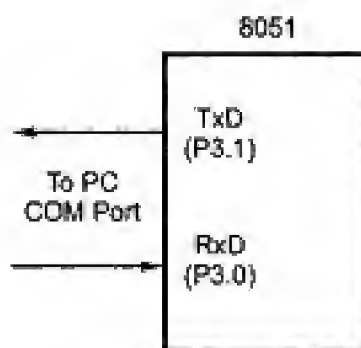


Fig. 10.5

```

MOV TMOD, #20H      ; Initialize timer 1 in mode 2
MOV TH1, #0FDH      ; Load count to get 9600 baud rate
MOV SCON, #50H      ; 8-bit, 1 stop, REN enabled
SETB TR1            ; Start timer 1
MOV DPTR, #2000H     ; Initialize memory pointer to
                    ; save received data
MOV R0, #05H        ; Initialize counter to read
                    ; 5 characters
RECV:  JNB RI, RECV   ; wait for character
MOV A, SBUF          ; Read the character
MOVX @DPTR, A        ; Save it in memory
INC DPTR             ; increment memory pointer
CLR RI              ; Get ready for next character
DJNZ R0, RECV        ; If not last character repeat
MOV DPTR, #MYDATA    ; Initialize pointer for message
CLR A
MOV R0, #4H          ; Initialize counter to send
                    ; 4 characters
MOVC A, @A+DPTR      ; Get the character
MOV SBUF, A          ; Load the data
HERE:  JNB TI, HERE   ; Wait for complete byte
                    ; transfer
CLR TI              ; get ready for next character
MYDATA: DB "Fine", 0
END

```

Review Questions

1. Draw and explain connection between RS-232C and 8051.
2. Explain operating modes for serial port in 8051.
3. Draw format of SCON register. Explain different bits in it.
4. Give steps to program 8051 for serial data transfer.
5. Give steps to program 8051 for receiving serial data.
6. How will you double the baud rate in 8051 ?



Interrupt Programming

In this chapter, we will discuss the interrupts supported by 8051 and related programming.

11.1 8051 interrupts

The 8051 provides five interrupt sources. These are listed below.

1. Timer 0 (TF0) and timer 1 (TF1) interrupt.
2. External hardware interrupts, $\overline{\text{INT0}}$ and $\overline{\text{INT1}}$.
3. Serial communication interrupt TI and RI

The Fig. 11.1 shows the interrupt structure of 8051.

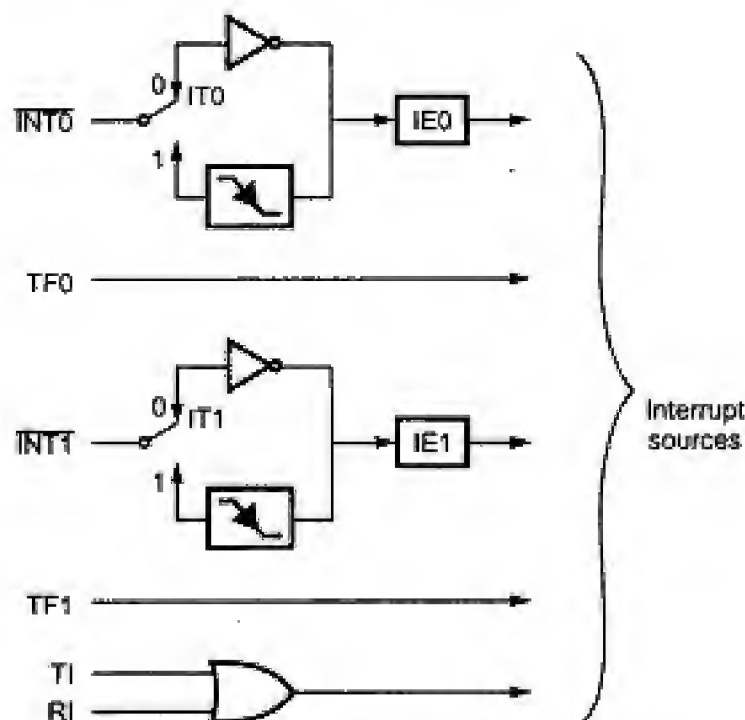


Fig. 11.1 MCS - 51 Interrupt structure

11.2 Interrupt Vector Table

In 8051, all interrupts are vectored interrupts and have vector locations as listed in Table 11.1 when interrupt is activated 8051 reads the address of interrupt service routine from the vector location.

| Interrupt | Vector Location |
|--|-----------------|
| External hardware interrupt 0 (INT0) | 0003H |
| Timer 0 interrupt (TF0) | 000BH |
| External hardware interrupt 1 (INT1) | 0013H |
| Timer 1 interrupt (TF1) | 001BH |
| Serial communication interrupt (RI and TI) | 0023H |

Table 11.1 Interrupt vector table for 8051

11.3 Enabling and Disabling an Interrupt

When 8051 is reset, all interrupts are disable. These are enabled by software. All of the bits that generate interrupts can be set or cleared by software, with the same result as though it had been set or cleared by hardware. That is, interrupts can be generated or pending interrupts can be cancelled in software.

Each of these interrupt sources can be individually enabled or disabled by setting or clearing a bit in Special Function Register IE (Fig. 11.2) . IE contains also a global disable bit, EA, which disables all interrupts at once.

Note in Fig. 11.2 that bit position IE.6 is unimplemented. In the 8051s, bit position IE.5 is also unimplemented. User software should not write 1s to these bit positions, since they may be used in future MCS-51 products.

| | | | | | | | |
|-------|---|-----|----|-------|-----|-----|-----|
| (MSB) | | | | (LSB) | | | |
| EA | - | ET2 | ES | ET1 | EX1 | ET0 | EX0 |

| Symbol | Position | Name and Significance |
|--------|----------|--|
| EA | IE.7 | Enable All control bit. Cleared by software to disable all interrupts, independent of the state of IE.4-IE.0. |
| - | IE.6 | (Reserved) |
| ET2 | IE.5 | (Reserved) |
| ES | IE.4 | Enable Serial port control bit. Set/cleared by software to enable/disable interrupts from TI or RI flags. |
| ET1 | IE.3 | Enable Timer 1 control bit. Set/cleared by software to enable/disable interrupts from timer/counter 1 |

| | | |
|-----|------|--|
| EX1 | IE.2 | Enable External interrupt 1 control bit. Set/cleared by software to enable/ disable interrupts from INT1. |
| ET0 | IE.1 | Enable Timer 0 control bit. Set/cleared by software to enable/disable interrupts from timer/counter 0. |
| EX0 | IE.0 | Enable external interrupt 0 control bit. Set/cleared by software to enable/disable interrupts from INTO. |

Fig. 11.2 IE-Interrupt enable register

11.4 Timer Interrupts and Programming

The Timer 0 and Timer 1 Interrupts are generated by TF0 and TF1, which are set by a rollover in their respective Timer/Counter registers (except see Timer 0 in Mode 3). When a timer interrupt is generated, the flag that generated it is cleared by the on-chip hardware when the service routine is vectored.

Recall the concepts studied in chapter 9. In that chapter we have seen the use of timer 0 and timer 1 with the polling method. Here, we are discussing the use of interrupts to program 8051 timers. We know that the timer flag (TF) is set (=1) when the timer rolls over. In polling method, the TF is monitored with the instruction 'JNB TF, target address'. We have to wait until the TF is raised. The problem with this polling method is that 8051 can not do anything else until TF is set to high. This problem can be solved using interrupt method. If the timer interrupt in the IE register is enabled, TF is set whenever the timer is rolled over and the 8051 is interrupted. Thus the 8051 can perform anything else until it is interrupted. After interruption (timer rolling over) only the 8051 remains busy in executing interrupt service routine.

➡ **Example 11.1 :** Write an 8051 ALP that continuously read 8-bit data from port2 and sends it to port 0. At the same time it should generate square wave of 500 μ s period on port 1.0. Assume the crystal frequency = 11.0592 MHz.

Solution : We will use timer 0 in autoreload mode, i.e. mode 2. To generate square wave of 500 μ s we have to toggle port 1.0 pin after every 250 μ s.

$$\text{Timer clock frequency} = \frac{11.0592 \times 10^6}{12} = 921.6 \text{ kHz}$$

$$\therefore \quad \text{TH0} = 256 - 250 \mu\text{s} \times 921.6 \times 10^3$$

$$= 26 = 1 \text{ AH}$$

Program :

```

ORG 0000H
LJMP MAIN          ; Avoid using memory space
                   ; allocated to interrupt vector table

ORG 000BH          ; ISR for Timer 0 interrupt
CPL P1.0           ; Complement P1.0 bit

```



```
        RETI                ; return from ISR

        ORG 0030H          ; Start main program after
                           ; interrupt vector table
MAIN :   MOV TMOD, #02H    ; Initialize timer 0 in mode 2
        MOV P2, #0FFH     ; Configure Port 2 as input
        MOV TH0, #AH      ; Load timer count
        MOV IE, #82H      ; Enable timer 0 interrupt
        SETB TR0          ; Start timer 0
BACK :   MOV A, P2         ; Read data from P2
        MOV P0, A         ; Send it on P0
        SJMP BACK        ; Repeat
        END
```

11.5 Programming External Hardware Interrupts

Pins, P 3.2 (pin number 12) and P 3.3 (pin number 13) in port 3 are used as external hardware interrupts INT0 and INT1, respectively. The external Interrupts $\overline{\text{INT0}}$ and $\overline{\text{INT1}}$ can each be either level-activated or transition-activated, depending on bits IT0 and IT1 in Register TCON. The flags that actually generate these interrupts are bits IE0 and IE1 in TCON. When an external interrupt is generated, the flag that generated it is cleared by the hardware when the service routine is vectored to only if the interrupt was transition-activated. If the interrupt was level-activated, then the external requesting source is what controls the request flag, rather than the on-chip hardware.

If ITx = 0, external interrupt x is triggered by a detected low at the $\overline{\text{INTx}}$ pin. If ITx = 1, external interrupt x is edge-triggered. In this mode if successive samples of the $\overline{\text{INTx}}$ pin show a high in one cycle and a low in the next cycle, interrupt request flag IEx in TCON is set. Flag bit IEx then requests the interrupt.

Since the external interrupt pins are sampled once each machine cycle, an input high or low should hold for at least 12 oscillator periods to ensure sampling. If the external interrupt is transition-activated, the external source has to hold the request pin high for at least one machine cycle, and then hold it low for at least one machine cycle to ensure that the transition is seen so that interrupt request flag IEx will be set. IEx will be automatically cleared by the CPU when the service routine is called.

If the external interrupt is level-activated, the external source has to hold the request active until the requested interrupt is actually generated. Then it has to deactivate the request before the interrupt service routine is completed, or else another interrupt will be generated.

➡ **Example 11.2 :** Write an 8051 ALP to glow LED for a fraction of second when external interrupt INT0 is activated.

Solution :

```
ORG 0000H
LJMP MAIN          ; Avoid using memory space
                   ; allocated to interrupt vector table

ORG 0003H
SETB P1.0          ; Turn ON LED
BACK :  MOVE R2, #0FFH ; Load count
        DJNZ BACK      ; Decrement count and if not zero repeat
        CLR P1.0       ; Turn OFF LED
        RETI           ; Return to main program

ORG 0030H          ; Start main program after interrupt
                   ; vector table
MAIN :  MOV IE, #10000001B ; Enable external interrupt 0
HERE :  SJMP HERE      ; wait for interrupt
END
```

11.6 Serial Communication Interrupts and Programming

The Serial port Interrupt is generated by the logical OR of RI and TI. Neither of these flags is cleared by hardware when the service routine is vectored to. In fact, the service routine will normally have to determine whether it was RI or TI that generated the interrupt, and the bit will have to be cleared in software.

Recall the serial communication studied in Chapter 10. In that chapter, the examples which we have seen, use polling method. Here, we are discussing interrupt based serial communication. In this case, the 8051 can perform other tasks in addition to serial communication, i.e. sending and receiving data from serial communication port.

We know from Chapter 10 that transmit interrupt (TI) flag is set (=1) when the last bit of the framed data (stop bit) is transmitted. This indicates that the SBUF register is ready to transmit the next byte. The receive interrupt (RI) flag is set (=1) when the complete frame of data (with stop bit) is received. RI indicates that the received byte needs to be picked up before it is lost by new incoming serial data.

All the above concepts are applied equally using polling or an interrupt. Only difference is in serving the serial communication needs. In polling method, the flag (TI or RI) is monitored. The 8051 can not do anything else until this flag is set to high. This problem is solved using interrupt method. When 8051 has received a byte or is ready to send the next byte, the RI or TI flag respectively is set. Any other work can be performed while the serial communication needs are served. There is a single interrupt set aside for serial communication. If IE register (IE.4) is enabled, when RI or TI is set (= 1), the 8051 is interrupted. When interrupted, the ISR written at 0023h is executed by 8051. In ISR, the TI

and RI flags must be examined to check which one caused the interrupt and according to flag the response is given.

➡ **Example 11.3 :** Write an 8051 ALP that continuously read 8-bit data from port 2 and sends it to port 0. At the same time it should read incoming data from serial port at baud rate 9600 and send it to port 1. Assume that crystal frequency = 11.0592 MHz.

Solution :

```

        ORG 0000H
        LJMP MAIN                ; Avoid using memory space
                                   ; allocated to interrupt vector table

        ORG 0023H
        JNB RI, SKIP             ; If RI is low goto skip.
        MOV A, SBUF              ; Otherwise receive serial data
        MOV P1, A                ; Send it to port 1
        CLR RI                   ; Clear RI
        RETI                     ; Return to main program
SKIP :   CLR TI                  ; Clear TI
        RETI                     ; Return to main program

        ORG 100H
MAIN :   MOV P2, #0FFH           ; Configure P2 as an input port
        MOV TMOD, #20H          ; Initialize timer 1 in mode 2
        MOV TH1, #FDH           ; Load count to get 9600 baud rate
        MOV SCON, #50H          ; Select serial mode with receiver
                                   ; enabled
        MOV IE, #10010000B      ; Enable serial interrupt
        SETB TR1                 ; Start timer 1
BACK :   MOV A, P2               ; Read data from port 2
        MOV P0, A               ; Send it to port 0
        SJMP BACK               ; Repeat
        END

```

➡ **Example 11.4 :** Write an 8051 ALP that continuously read 8-bit data from port 2 and sends it to port 0. At the same time it should transmit the same data on serial port. Assume that crystal frequency = 11.0592 MHz.

Solution :

```

        ORG 0000H
        LJMP MAIN                ; Avoid using memory space
                                   ; allocated to interrupt vector table

        ORG 0023H
        JNB TI, SKIP             ; If TI is low goto SKIP
        MOV SBUF, A              ; Transfer data serially
        CLR TI                   ; Clear TI
        RETI                     ; Return to main program
SKIP :   CLR RI                  ; Clear RI
        RETI                     ; Return to main program

```

```

      ORG 100H
MAIN : MOV P2, 0FFH      ; Configure P2 as an input port
      MOV TMOD, #20H     ; Initialize timer 1 in mode 2
      MOV TH1, #FDH      ; Load count to get 9600 baud rate
      MOV SCON, #40H     ; Select serial mode
      MOV IE, #10010000B ; Enable serial interrupt
      SETB TRI           ; Start timer 1
      MOV A, P2          ; Read data from port 2
      MOV SBUF, A        ; Send the first byte serially
BACK : MOV A, P2          ; Read data from port 2
      MOV P0, A          ; Send it to port 0
      SJMP BACK          ; Repeat
      END

```

11.7 Interrupt Priority in the 8051

Each interrupt source can also be individually programmed to one of two priority levels by setting or clearing a bit in Special Function Register IP (Fig. 11.3). A low-priority interrupt can itself be interrupted by a high-priority interrupt, but not by another low priority interrupt. A high-priority interrupt can't be interrupted by any other interrupt source.


| (MSB) | | | | (LSB) | | | |
|-------|---|---|----|-------|-----|-----|-----|
| - | - | - | PS | PT1 | PX1 | PT0 | PX0 |

| Symbol | Position | Name and Significance |
|--------|----------|---|
| - | IP.7 | (Reserved) |
| - | IP.6 | (Reserved) |
| - | IP.5 | (Reserved) |
| PS | IP.4 | Serial port Priority control bit. Set/cleared by software to specify high/low priority interrupts for Serial port. |
| PT1 | IP.3 | Timer 1 Priority control bit Set/cleared by software to specify high/low priority interrupts for timer/counter 1. |
| PX1 | IP.2 | External interrupt 1 Priority control bit. Set/cleared by software to specify high/low priority interrupts for INT1. |
| PT0 | IP.1 | Timer 0 Priority control bit. Set/cleared by software to specify high/low priority interrupts for timer/counter 0. |
| PX0 | IP.0 | External interrupt 0 Priority control bit. Set/cleared by software to specify high/low priority interrupts for INT0. |

Fig. 11.3 IP - Interrupt priority control register

If two requests of different priority levels are received simultaneously, the request of higher priority level is served. If requests of the same priority level are received

simultaneously, an internal polling sequence determines which request is serviced. Thus within each priority level there is a second priority structure determined by the polling sequence, as follows :

| No. | Source | Priority within Level |
|-----|---------|---|
| 1. | IE0 | (highest) |
| 2. | TF0 |  |
| 3. | IE1 | |
| 4. | TF1 | |
| 5. | RI + TI | (lowest) |

Note that the "priority within level" structure is only used to resolve simultaneous requests of the same priority level.

The IP register contains a number of unimplemented bits. IP.7 and IP.6 are vacant in the 8052s, and in the 8051s these and IP.5 are vacant. User software should not write 1s to these bit positions, since they may be used in future MCS-51 products.

11.7.1 Nested Interrupts

Consider a case, the 8051 is executing an ISR for servicing an interrupt and another interrupt is occurred. In such a case, if the new coming interrupt is high priority interrupt then only it can interrupt the previously occurred low-priority interrupt. These are called 'nested interrupts'. Thus, in 8051 a low-priority interrupt can be interrupted by high-priority interrupt but not by another low-priority interrupt. In 8051, all the interrupts are latched and kept internally. But the low-priority interrupt is serviced only after finishing the servicing of the high-priority interrupts.

11.7.2 Software Triggering of Interrupt

Software triggering of the interrupts is possible in 8051. This means the interrupt can be caused by setting an interrupt flag with an instruction. For example, if the IE bit for timer 1 is set, an instruction 'SETB TF1' will interrupt the 8051 and 8051 will start executing ISR. Thus, it is not needed to wait for timer 1 to roll over to have an interrupt. Since we are using instruction to create an interrupt, it is called software triggering. This is useful for testing an ISR by way of simulation.

Review Questions

1. Draw and explain the interrupt structure of 8051.
2. When 8051 is reset, all interrupts are disabled. How to enable these interrupts ?
3. Give the vector addresses of each interrupt source.
4. Explain, how to enable and disable interrupts.
5. With the help of example explain programming of timer interrupt.

6. *With the help of example explain programming of external interrupts.*
7. *With the help of example explain programming of serial interrupt.*
8. *Write a short note on interrupt priority.*
9. *What are nested interrupts ?*
10. *What do you mean by software triggering of interrupts ? Whether is it possible in 8051 ?*

□□□

Programmable Peripheral Interface-8255

The 8255 is a general purpose programmable I/O device used for parallel data transfer. It has 24 I/O pins which can be grouped in three 8-bit parallel ports : Port A, Port B and Port C. The eight bits of port C can be used as individual bits or be grouped in two 4-bit ports : C_{upper} (C_U) and C_{lower} (C_L).

The 8255, primarily, can be programmed in two basic modes : Bit Set/Reset (BSR) mode and I/O mode. The BSR mode is used to set or reset the bits in port C. The I/O mode is further divided into three modes :

Mode 0 : Simple Input/Output

Mode 1 : Input/Output with handshake

Mode 2 : Bi-directional I/O data transfer

The function of I/O pins (input or output) and modes of operation of I/O ports can be programmed by writing proper control word in the control word register. Each bit in the control word has a specific meaning and the status of these bits decides the function and operating mode of the I/O ports.

12.1 Features of 8255A

1. The 8255A is a widely used, programmable, parallel I/O device.
2. It can be programmed to transfer data under various conditions, from simple I/O to interrupt I/O.
3. It is compatible with all Intel and most other microprocessors.
4. It is completely TTL compatible.
5. It has three 8-bit ports : Port A, Port B, and Port C, which are arranged in two groups of 12 pins. Each port has an unique address, and data can be read from or written to a port. In addition to the address assigned to the three ports, another address is assigned to the control register into which control words are written for programming the 8255 to operate in various modes.
6. Its bit set/reset mode allows setting and resetting of individual bits of Port C.
7. The 8255 can operate in 3 I/O modes : (i) Mode 0, (ii) Mode 1, and (iii) Mode 2.

- a) In Mode 0, Port A and Port B can be configured as simple 8-bit input or output ports without handshaking. The two halves of Port C can be programmed separately as 4-bit input or output ports.
 - b) In Mode 1, two groups each of 12 pins are formed. Group A consists of Port A and the upper half of Port C while Group B consists of Port B and the lower half of Port C. Ports A and B can be programmed as 8-bit Input or Output ports with three lines of Port C in each group used for handshaking.
 - c) In Mode 2, only Port A can be used as a bidirectional port. The handshaking signals are provided on five lines of Port C ($PC_3 - PC_7$). Port B can be used in Mode 0 or in Mode 1.
8. All I/O pins of 8255 has 2.5 mA DC driving capacity (i.e. sourcing current of 2.5 mA).

12.2 Pin Diagram

Fig. 12.1 shows the pin diagram of 8255.

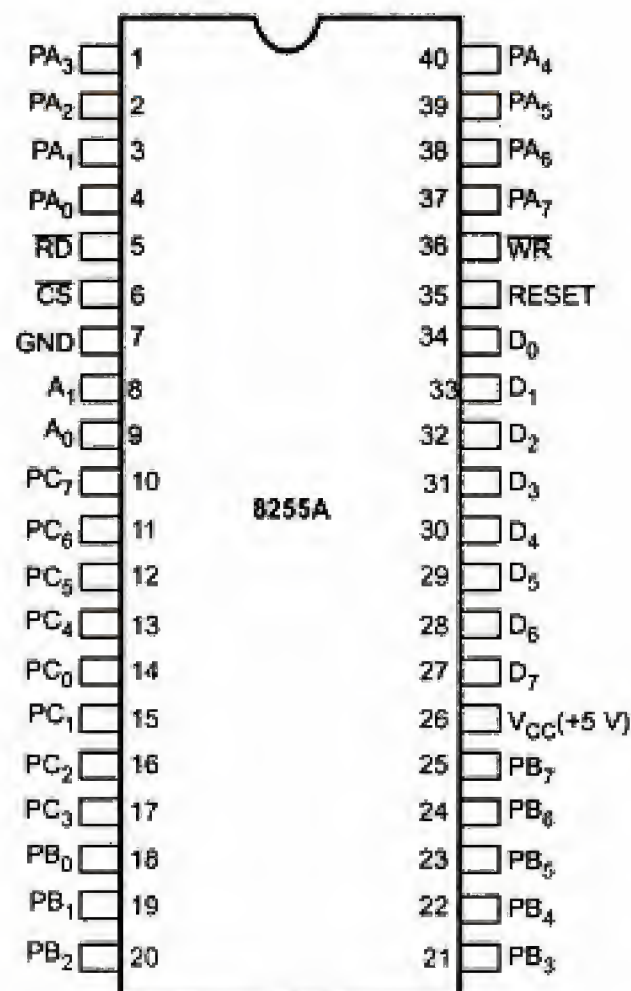


Fig. 12.1 Pin diagram of 8255A

| Pin Symbols | Function |
|-------------------------------|--|
| D_0-D_7 (Data Bus) | These bi-directional, tri-state data bus lines are connected to the system data bus. They are used to transfer data and control word from microprocessor (8085) to 8255 or to receive data or status word from 8255 to the 8085. |
| PA_0-PA_7 (Port A) | These 8-bit bi-directional I/O pins are used to send data to output device and to receive data from input device. It functions as an 8-bit data output latch/buffer, when used in output mode and an 8-bit data input buffer, when used in input mode. |
| PB_0-PB_7 (Port B) | These 8-bit bi-directional I/O pins are used to send data to output device and to receive data from input device. It functions as an 8-bit data, output latch/buffer when used in output mode and an 8-bit data input buffer, when used in input mode. |
| PC_0-PC_7 | These 8-bit bi-directional I/O pins are divided into two groups PC_L (PC_3-PC_0) and PC_U (PC_7-PC_4). These groups individually can transfer data in or out when programmed for simple I/O, and used as handshake signals when programmed for handshake or bi-directional modes. |
| \overline{RD} (Read) | When this pin is low, the CPU can read the data in the ports or the status word, through the data buffer. |
| \overline{WR} (Write) | When this input pin is low, the CPU can write data on the ports or in the control register through the data bus buffer. |
| \overline{CS} (Chip Select) | This is an active low input which can be enabled for data transfer operation between the CPU and the 8255. |
| RESET | This is an active high input used to reset 8255. When RESET input is high, the control register is cleared and all the ports are set to the input mode. Usually RESET OUT signal from 8085 is used to reset 8255. |
| A_0 and A_1 | These input signals along with \overline{RD} and \overline{WR} inputs control the selection of the control/status word registers or one of the three ports. Table. 5.1 summarizes the status of A_0 , A_1 , \overline{CS} , \overline{RD} and \overline{WR} to access the control word/ports. A_0 and A_1 are generally connected to the A_0 , A_1 pins of the address bus; the 8255 therefore occupies four consecutive locations in the I/O space. |

| A_1 | A_0 | \overline{RD} | \overline{WR} | \overline{CS} | Operations |
|-------|-------|-----------------|-----------------|-----------------|---------------------------------|
| | | | | | Input (Read) Operation |
| 0 | 0 | 0 | 1 | 0 | Port A to Data Bus |
| 0 | 1 | 0 | 1 | 0 | Port B to Data Bus |
| 1 | 0 | 0 | 1 | 0 | Port C to Data Bus |
| | | | | | Output (Write) Operation |
| 0 | 0 | 1 | 0 | 0 | Data Bus to Port A |
| 0 | 1 | 1 | 0 | 0 | Data Bus to Port B |
| 1 | 0 | 1 | 0 | 0 | Data Bus to Port C |
| 1 | 1 | 1 | 0 | 0 | Data Bus to Control Register |

| | | | | | Disable Function |
|---|---|---|---|---|---------------------|
| | | | | | |
| | | | | | |
| | | | | | |
| X | X | X | X | 1 | Data Bus Tri-stated |
| 1 | 1 | 0 | 1 | 0 | Illegal Condition |
| X | X | 1 | 1 | 0 | Data Bus Tri-stated |

Table 12.1 Port and register select signals summary

12.3 Block Diagram

Fig. 12.2 shows the internal block diagram of 8255A. It consists of data bus buffer, control logic and Group A and Group B controls.

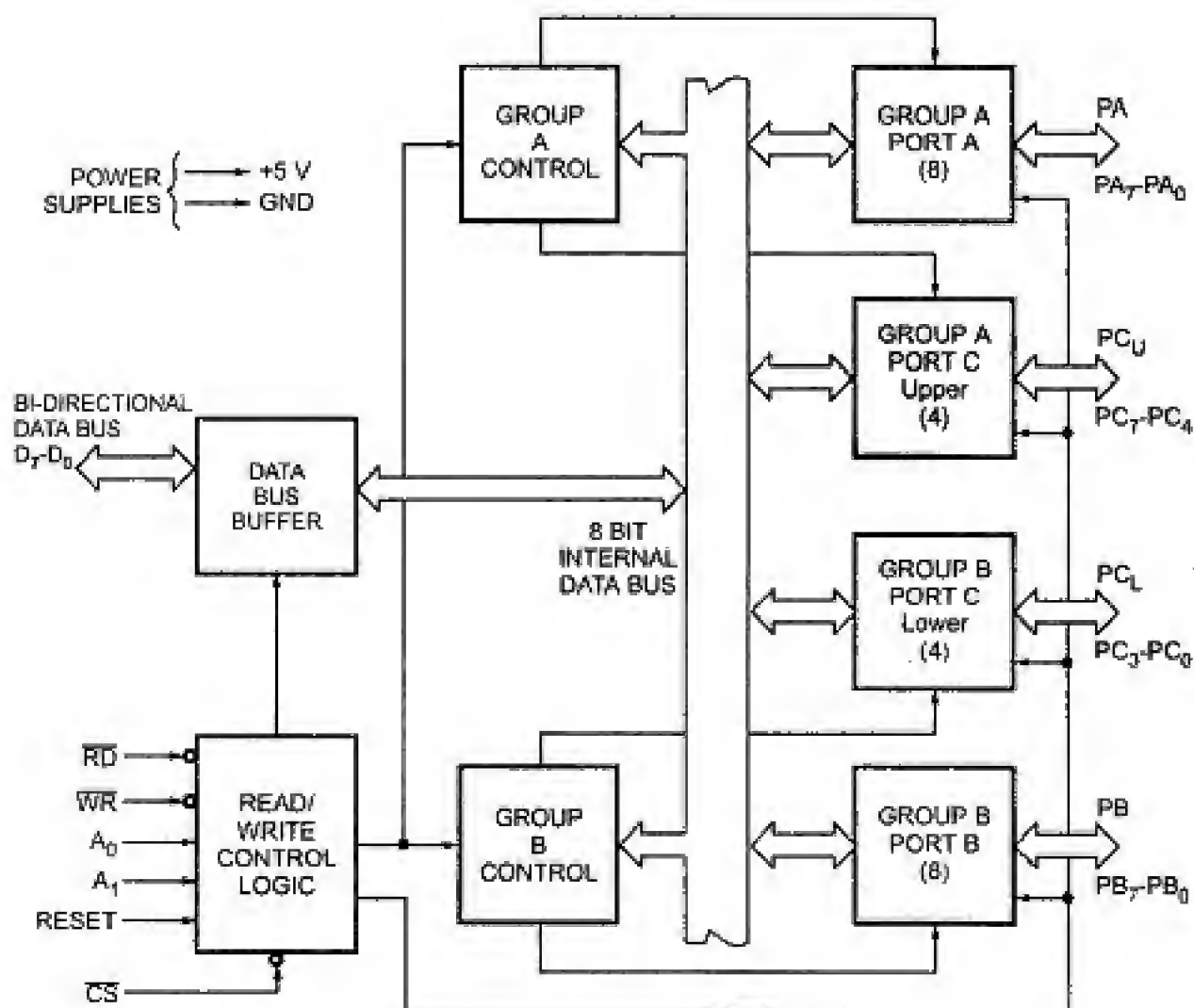


Fig. 12.2 Block diagram of 8255A

12.3.1 Data Bus Buffer

This tri-state bi-directional buffer is used to interface the internal data bus of 8255 to the system data bus. Input or Output instructions executed by the CPU either Read data from, or Write data into the buffer. Output data from the CPU to the ports or control register, and input data to the CPU from the ports or status register are all passed through the buffer.

12.3.2 Control Logic

The control logic block accepts control bus signals as well as inputs from the address bus, and issues commands to the individual group control blocks (Group A control and Group B control). It issues appropriate enabling signals to access the required data/control words or status word. The input pins for the control logic section are described here.

12.3.3 Group A and Group B Controls

Each of the Group A and Group B control blocks receives control words from the CPU and issues appropriate commands to the ports associated with it. The Group A control block controls Port A and PC₇ - PC₄ while the Group B control block controls Port B and PC₃-PC₀.

- Port A :** This has an 8-bit latched and buffered output and an 8-bit input latch. It can be programmed in three modes: mode 0, mode 1 and mode 2.
- Port B :** This has an 8-bit data I/O latch/buffer and an 8-bit data input buffer. It can be programmed in mode 0 and mode 1.
- Port C :** This has one 8-bit unlatched input buffer and an 8-bit output latch/buffer. Port C can be separated into two parts and each can be used as control signals for ports A and B in the handshake mode. It can be programmed for bit set/reset operation.

12.4 Operation Modes

12.4.1 Bit Set-Reset (BSR) Mode

The individual bits of Port C can be set or reset by sending out a single OUT instruction to the control register. When Port C is used for control/status operation, this feature can be used to set or reset individual bits.

12.4.2 I/O Modes

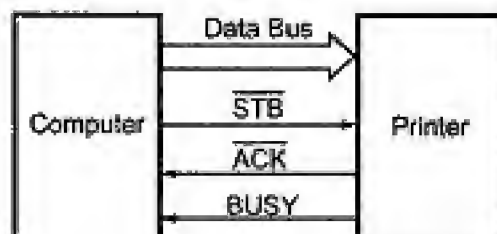
Mode 0 : Simple Input/output

In this mode, ports A and B are used as two simple 8-bit I/O ports and Port C as two 4-bit ports. Each port (or half - port, in case of C) can be programmed to function as simply an input port or an output port. The input/output features in Mode 0 are as follows :

1. Outputs are latched.
2. Inputs are buffered, not latched.
3. Ports do not have handshake or interrupt capability.

Mode 1 : Input/Output with handshake

In this mode, input or output data transfer is controlled by handshaking signals. Handshaking signals are used to transfer data between devices whose data transfer speeds



are not same. For example, computer can send data to the printer with large speed but printer can't accept data and print data with this rate. So computer has to send data with the speed with which printer can accept. This type of data transfer is achieved by using handshaking signals alongwith data signals. Fig. 12.3 shows data transfer between computer and printer using handshaking signals.

Fig. 12.3 Data transfer between computer and printer using handshaking signals

These handshaking signals are used to tell computer whether printer is ready to accept the data or not. If printer is ready to accept the data then after sending data on data bus, computer uses another handshaking signal (\overline{STB}) to tell printer that valid data is available on the data bus.

The 8255 mode 1 which supports handshaking has following features.

1. Two ports (A and B) function as 8-bit I/O ports. They can be configured either as input or output ports.
2. Each port uses three lines from Port C as handshake signals. The remaining two lines of Port C can be used for simple I/O functions.
3. Input and output data are latched.
4. Interrupt logic is supported.

Mode 2 : Bi-directional I/O data transfer

This mode allows bi-directional data transfer (transmission and reception) over a single 8-bit data bus using handshaking signals. This feature is available only in Group A with Port A as the 8-bit bidirectional data bus; and PC_3 - PC_7 are used for handshaking purpose. In this mode, both inputs and outputs are latched. Due to use of a single 8-bit data bus for bi-directional data transfer, the data sent out by the CPU through Port A

appears on the bus connecting it to the peripheral, only when the peripheral requests it. The remaining lines of Port C i.e. PC_0 - PC_2 can be used for simple I/O functions. The Port B can be programmed in mode 0 or in mode 1. When Port B is programmed in mode 1, PC_0 - PC_2 lines of Port C are used as handshaking signals.

12.5 Control Word Formats

A high on the RESET pin causes all 24 lines of the three 8-bit ports to be in the input mode. All flip-flops are cleared and the interrupts are reset. This condition is maintained even after the RESET goes low. The ports of the 8255 can then be programmed for any other mode by writing a single control word into the control register, when required.

12.5.1 For Bit Set/Reset Mode

Fig. 12.4 shows bit set/reset control word format.

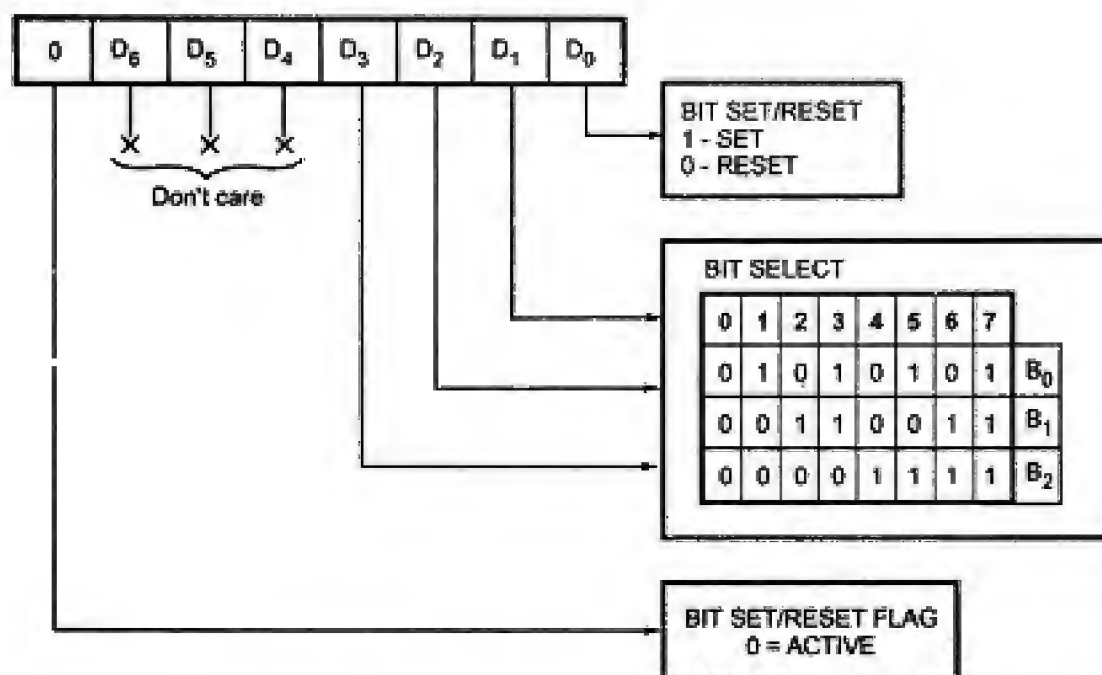


Fig. 12.4 Bit set/reset control word format

The eight possible combinations of the states of bits D_3 - D_1 (B_2 B_1 B_0) in the Bit Set-Reset format (BSR) determine particular bit in PC_0 - PC_7 being set or reset as per the status of bit D_0 . A BSR word is to be written for each bit that is to be set or reset. For example, if bit PC_3 is to be set and bit PC_4 is to be reset, the appropriate BSR words that will have to be loaded into the control register will be, $0XXX0111$ and $0XXX1000$, respectively, where X is don't care.

The BSR word can also be used for enabling or disabling interrupt signals generated by Port C when the 8255 is programmed for Mode 1 or 2 operation. This is done by setting or resetting the associated bits of the interrupts. This is described in detail in next section.

12.5.2 For I/O Mode

The mode definition format for I/O mode is shown in Fig. 12.5. The control words for both, mode definition and Bit Set-Reset are loaded into the same control register, with bit D_7 used for specifying whether the word loaded into the control register is a mode definition word or Bit Set-Reset word. If D_7 is high, the word is taken as a mode definition word, and if it is low, it is taken as a Bit Set-Reset word. The appropriate bits are set or reset depending on the type of operation desired, and loaded into the control register.

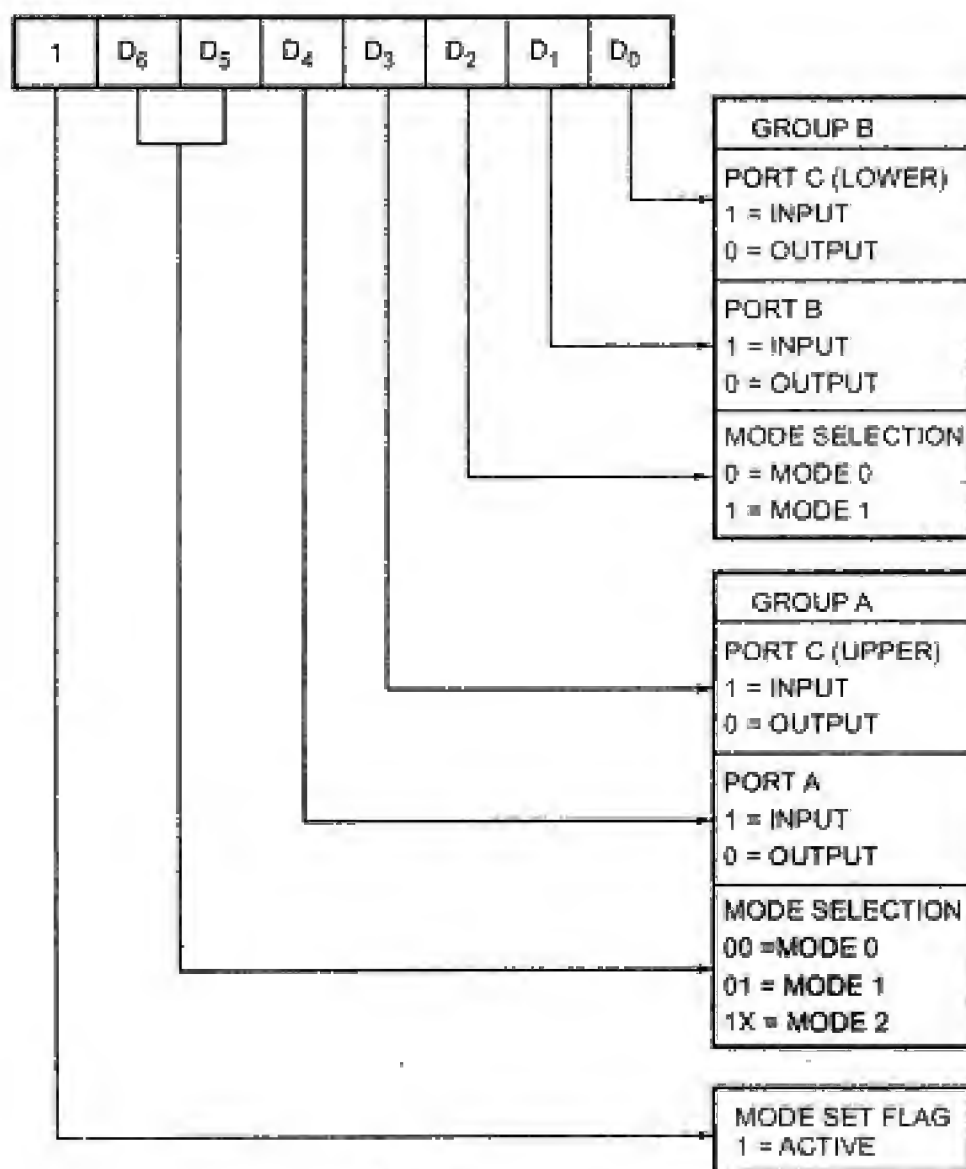


Fig. 12.5 8255 Mode definition format

➡ **Example 12.1 :** Write a program to initialize 8255 in the configuration given below :

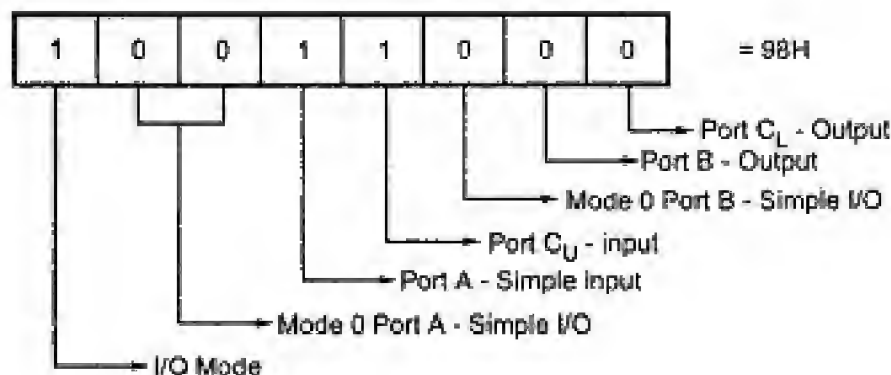
1. Port A : Simple input
2. Port B : Simple output
3. Port C_L : Output
4. Port C_U : Input

Assume address of the control word register of 8255 as 83H.

Solution :

Source program :

```
MVI A, 98H      ; Load control word
OUT 83H         ; Send control word
```

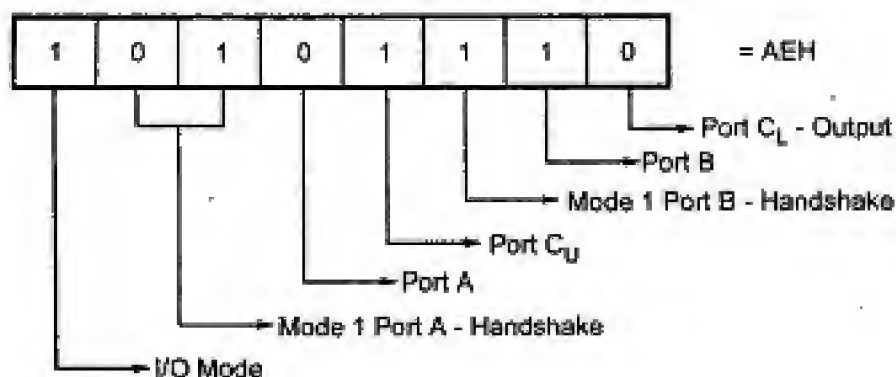


➡ **Example 12.2 :** Write a program to initialize 8255 in the configuration given below :

1. Port A : Output with handshake
2. Port B : Input with handshake
3. Port C_L : Output
4. Port C_U : Input

Assume address of the control word register of 8255 as 23H.

Solution :



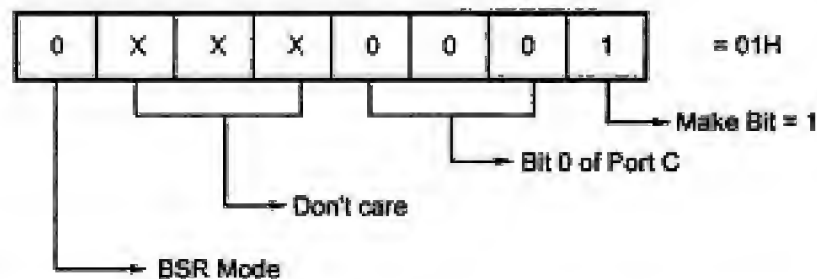
Source program :

```
MVI A, AEH      ; Load control word
OUT 23H         ; Send control word
```

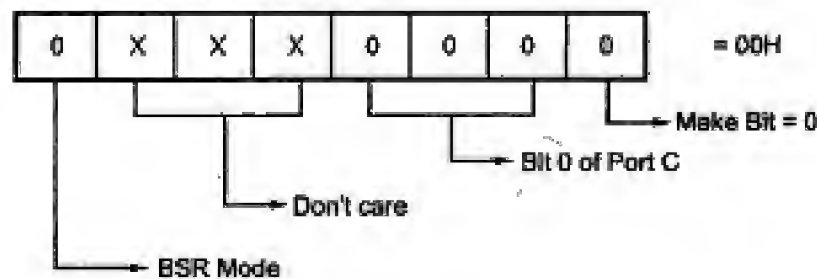
Lab Experiment 62 : Blink port C bit 0 of 8255.

Statement : Write a program to blink Port C bit 0 of the 8255. Assume address of control word register of 8255 as 83H. Use Bit Set/Reset mode.

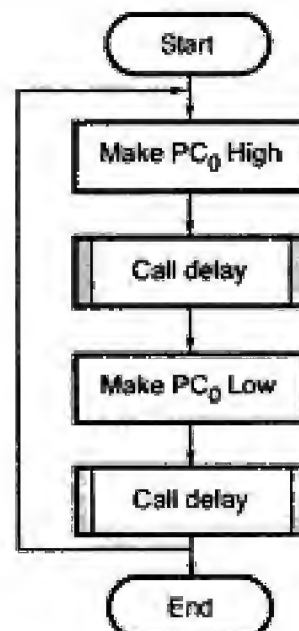
Solution : Control word to make bit 0 high.



Control word to make Bit 0 low



Flowchart :



Source program :

```

BACK :   MVI A, 01H    ; Load bit pattern to make PC0 high
        OUT 83H        ; Send it to control word register
        CALL DELAY     ; Call Delay subroutine
        MVI A, 00H     ; Load bit pattern to make PC0 Low
        OUT 83H        ; Send it to control word register
        CALL Delay      ; Call Delay subroutine
        JMP BACK       ; Repeat

```

12.6 8255 Programming and Operation**12.6.1 Programming in Mode 0**

The Ports A, B and C can be configured as simple input or output ports by writing the appropriate control word in the control word register. In the control word, D_7 is set to '1' (to define a mode set operation) and D_6 , D_5 and D_2 are all set to '0' to configure all the ports in Mode 0 operation. The status of bits D_4 , D_3 , D_1 and D_0 then determine (refer to Fig. 12.5) whether the corresponding ports are to be configured as Input or Output.

For example in mode 0, if Port A and Port B are to operate as output ports with Port C lower as input, and Port C upper as output, the control word that will have to be loaded into the control register will be as follows.

| D_7 | D_6 | D_5 | D_4 | D_3 | D_2 | D_1 | D_0 | |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | = 81H |

As mentioned earlier, this mode provides simple input and output operations for each of the three ports. No handshaking is required, data is simply written to or read from a specified port.

Input Mode : Fig. 12.6 shows the timing diagram for mode 0 input mode.

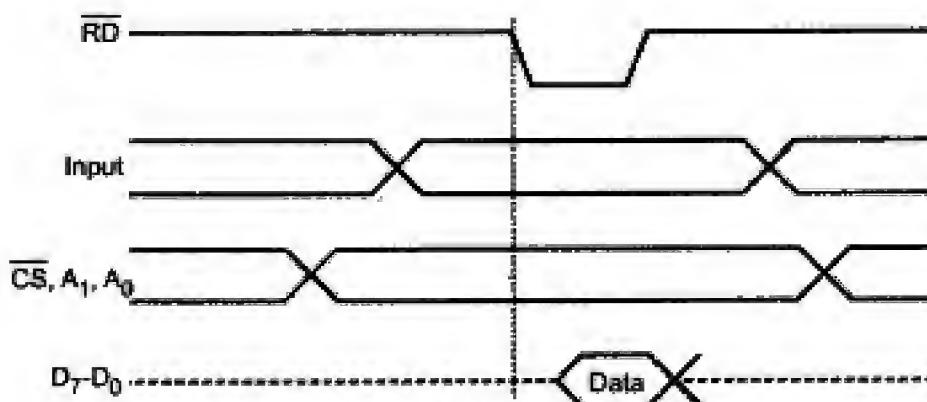


Fig. 12.6 Timing diagram for mode 0 input mode

After initialization of 8255 in the input mode 0, CPU can read data through the input port by initiating read command with proper port address. Read command activates \overline{RD} signal. Upon activation of \overline{RD} signal CPU reads the data from the selected input port into the CPU register.

Output Mode : Fig. 12.7 shows the timing diagram for mode 0 output mode.

After initialization of 8255 in the output mode 0, CPU can write data into the output port by initiating write command with proper port address. CPU sends data on the data bus and upon activation of \overline{WR} signal, data on the data bus gets latched on the selected output port.

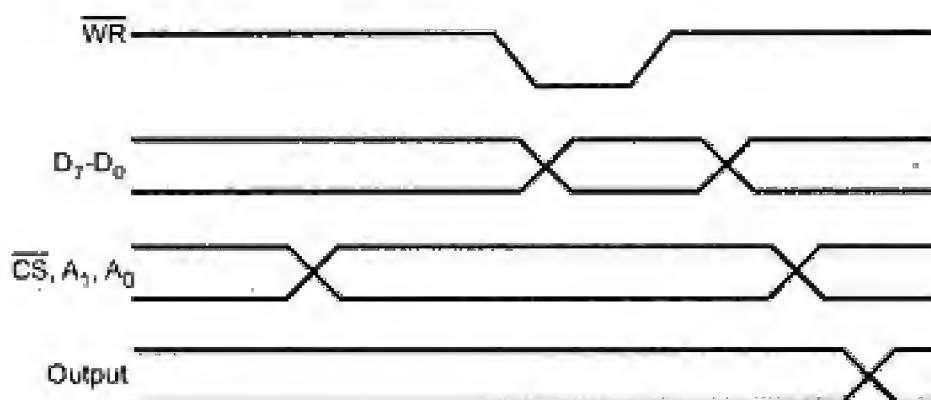


Fig. 12.7 Timing diagram for mode 0 output mode

Mode 0 Configurations :

| A | | B | | GROUP A | | | GROUP B | |
|-------|-------|-------|-------|---------|----------------|---|---------|----------------|
| D_4 | D_3 | D_1 | D_0 | PORT A | PORT C (Upper) | # | PORT B | PORT C (Lower) |
| 0 | 0 | 0 | 0 | OUTPUT | OUTPUT | 0 | OUTPUT | OUTPUT |
| 0 | 0 | 0 | 1 | OUTPUT | OUTPUT | 1 | OUTPUT | INPUT |
| 0 | 0 | 1 | 0 | OUTPUT | OUTPUT | 2 | INPUT | OUTPUT |
| 0 | 0 | 1 | 1 | OUTPUT | OUTPUT | 3 | INPUT | INPUT |
| 0 | 1 | 0 | 0 | OUTPUT | INPUT | 4 | OUTPUT | OUTPUT |
| 0 | 1 | 0 | 1 | OUTPUT | INPUT | 5 | OUTPUT | INPUT |
| 0 | 1 | 1 | 0 | OUTPUT | INPUT | 6 | INPUT | OUTPUT |
| 0 | 1 | 1 | 1 | OUTPUT | INPUT | 7 | INPUT | INPUT |

| | | | | | | | | |
|---|---|---|---|-------|--------|----|--------|--------|
| 1 | 0 | 0 | 0 | INPUT | OUTPUT | 8 | OUTPUT | OUTPUT |
| 1 | 0 | 0 | 1 | INPUT | OUTPUT | 9 | OUTPUT | INPUT |
| 1 | 0 | 1 | 0 | INPUT | OUTPUT | 10 | INPUT | OUTPUT |
| 1 | 0 | 1 | 1 | INPUT | OUTPUT | 11 | INPUT | INPUT |
| 1 | 1 | 0 | 0 | INPUT | INPUT | 12 | OUTPUT | OUTPUT |
| 1 | 1 | 0 | 1 | INPUT | INPUT | 13 | OUTPUT | INPUT |
| 1 | 1 | 1 | 0 | INPUT | INPUT | 14 | INPUT | OUTPUT |
| 1 | 1 | 1 | 1 | INPUT | INPUT | 15 | INPUT | INPUT |

12.6.2 Programming in Mode 1 (Input / Output with Handshake)

Both Group A and Group B can operate in Mode 1, either together, or individually, with each port containing an 8-bit latched Input or Output data port, and a 4-bit port which is used for control and status of the 8-bit port.

When Port A is to be programmed as an input port, PC_3 , PC_4 and PC_5 are used for control. PC_6 and PC_7 are not used and can be Input or Output, as programmed by bit D_3 of the control word. When Port A is programmed as an output port, PC_3 , PC_6 , and PC_7 are used for control and PC_4 and PC_5 can be Input or Output, as programmed by bit D_3 of the control word.

When port B is to be programmed as an input or output port, PC_0 , PC_1 and PC_2 are used for control.

Mode 1 Input control signals :

- 1. \overline{STB} (Strobe Input) :** This is an active low input signal for 8255 and output signal for the input device. The input device activates this signal to indicate CPU that the data to be read is already sent on the port lines of 8255 port. Upon activation of this signal 8255 loads the data from the input port lines into the input buffer of that port.
- 2. IBF (Input Buffer Full) :** This is an active high output signal for 8255 and an input signal for input device. This signal is generated by 8255 in response to \overline{STB} signal as an acknowledgement to input device. It also indicates to the input device that the input buffer is full and it is not ready to accept next byte from the input device. Therefore input device sends data on the port lines only when IBF signal is not active. The IBF signal is deactivated when CPU reads the data from input buffer of the respective port by activation of \overline{RD} signal.
- 3. INTR (Interrupt Request) :** This is an active high output signal generated by 8255. A 'high' on this output can be used to interrupt the CPU when an input device is requesting service. The 8255 sets the INTR when \overline{STB} signal is 'one', IBF signal is 'one' and INTE is 'one', indicating CPU that the data from the input device is available in the input buffer.

This signal is reset by the falling edge of the \overline{RD} signal i.e. immediately after reading the data from the input buffer.

INTE (Interrupt Enable) flip flop is used to enable or disable INTR (Interrupt request) signal. If INTE flip-flop is set, the interrupt request is generated depending on the status of \overline{STB}_A and IBF_A signals. If INTE flip flop is reset, the interrupt request is not generated, allowing masking facility for the interrupt.

Mode 1 : Port A Input operation

Fig. 12.8 (a) shows Port A as an input port along with the control word and control signals (for handshaking with a peripheral). When the control word (as in Fig. 12.8 (a)) is loaded into the control register, Group A is configured in Mode 1 with Port A as an input port. Port A can accept parallel data from a peripheral (like a keyboard) and this data can be read by the CPU. The peripheral first loads data into Port by making the \overline{STB}_A input low. This latches the data placed by the peripheral on the common data bus into Port A. Port A acknowledges reception of data by making IBF_A (Input Buffer Full) high. IBF_A is set when the \overline{STB}_A input is made low, as shown in Fig. 12.8 (b).

$INTR_A$ is an active high output signal which can be used to interrupt the CPU so that

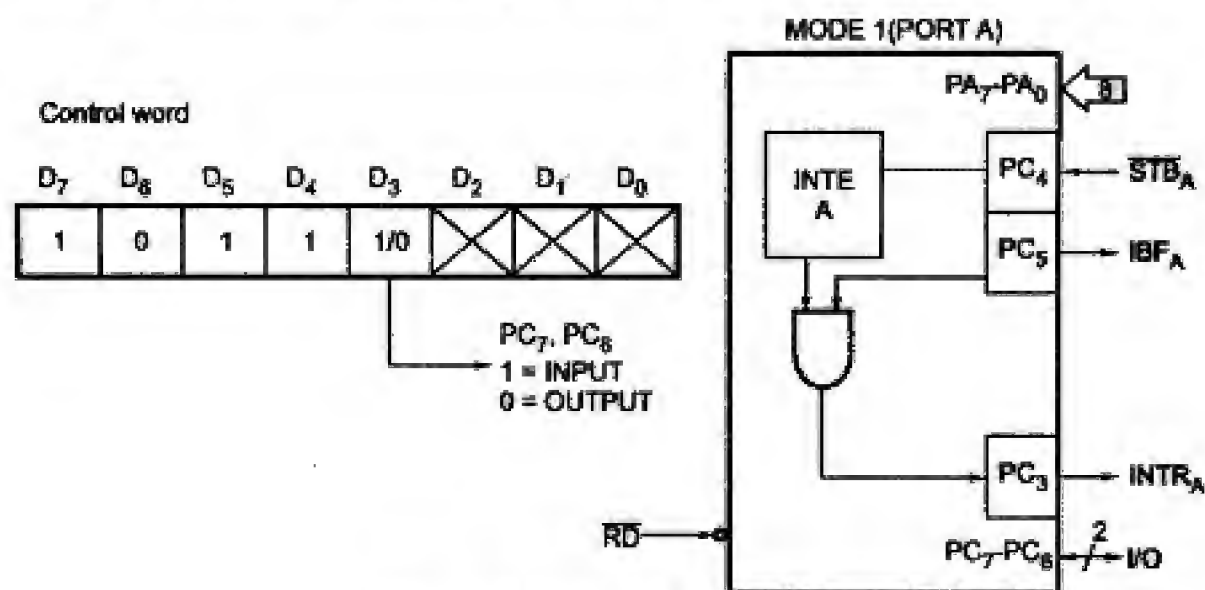


Fig. 12.8 (a) Port A in mode 1

the CPU can suspend its current operation and read the data written into Port A by the peripheral. $INTR_A$ can be enabled or disabled by the $INTE_A$ flip-flop which is controlled by Bit Set-Reset operation of PC_4 . $INTR_A$ is set (if enabled by setting the $INTE_A$ flip-flop) after the \overline{STB}_A has gone high again, and if IBF_A is high.

On receipt of the interrupt, the CPU can be forced to read Port A. The falling edge of the \overline{RD} input resets IBF_A and it goes low. This can be used to indicate to the peripheral that the input buffer is empty and that data can again be loaded into it.

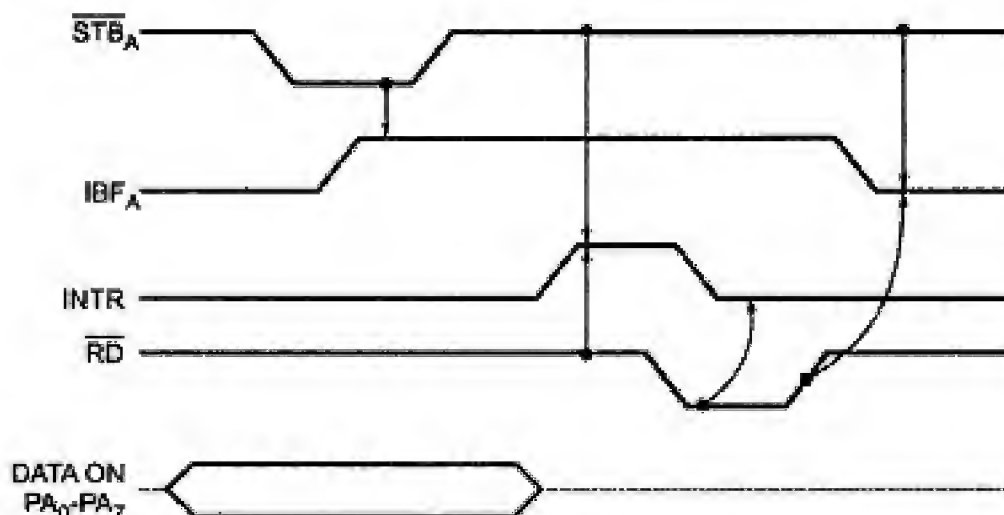


Fig. 12.8 (b) Timing diagram for port A in mode 1

Mode 1 : Port B input operation.

Fig. 12.9 shows Port B as an input port (when in Mode 1). The timing diagram and operation of Port B is similar to that of Port A except that it uses different bits of Port C for control. $INTE_B$ is controlled by Bit Set/Reset of PC_2 .

If the CPU is busy with other system operations, it can read data from the input port when it is interrupted. This is often called Interrupt driven I/O. However, if the CPU is otherwise not busy with other jobs, it can continuously poll (read) the status word to

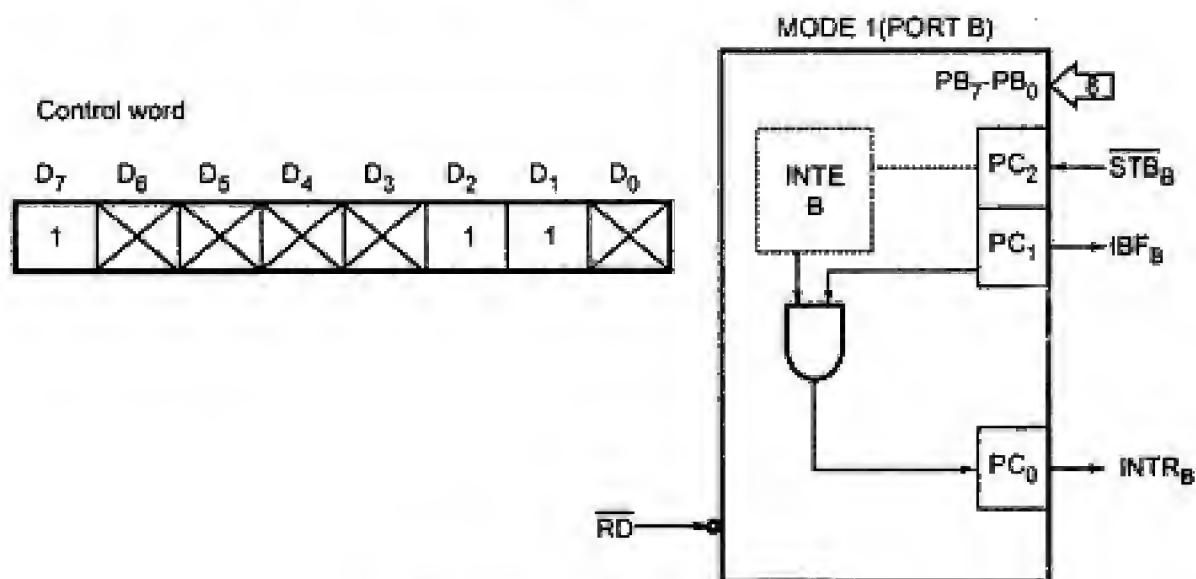


Fig. 12.9 Port B in mode 1

check for an IBF_A . This is often called Program Controlled I/O. The status word is accessed by reading Port C (A_1A_0 must be 10, \overline{RD} and \overline{CS} must be low). The status word format when Ports A and B are input ports in Mode 1, is shown in Fig. 12.10.

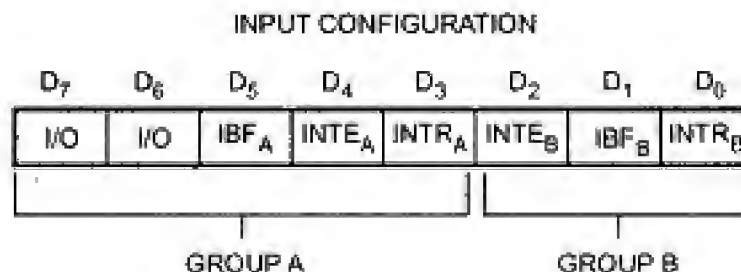


Fig. 12.10 Mode 1 status word (Input)

Mode 1 : Output control signals

1. \overline{OBF} (Output Buffer Full) :

This is an active low output signal for 8255 and input signal for the output device. The 8255 activates this signal to indicate output device that data is available on the output port. Upon activation of \overline{OBF} signal, output device reads data from the output port and acknowledges it by \overline{ACK} signal. The \overline{OBF} signal is activated at the rising edge of the \overline{WR} signal and de-activated at the falling edge of the \overline{ACK} signal.

2. \overline{ACK} (Acknowledge Input) :

This is an active low input signal for 8255 and output signal for the output device. The output device generates this signal to indicate 8255 that the data from port A or Port B has been accepted.

3. $INTR$ (Interrupt Request) :

This is an active high output signal generated by 8255. A 'high' on this output can be used to interrupt the CPU when an output device has accepted data transmitted by the CPU. The 8255 sets the $INTR$ when \overline{ACK} signal is 'one', \overline{OBF} is 'one' and $INTE$ is 'one', indicating that the output device is ready to accept next data byte. This signal is reset by the falling edge of the \overline{WR} signal i.e. immediately after sending the data to the output port.

$INTE$ (Interrupt Enable) flip flop is used to enable or disable $INTR$ (Interrupt Request) signal. If $INTE$ flip flop is set, the interrupt request is generated depending on the status of \overline{ACK} and \overline{OBF} signals. If $INTE$ flip flop is reset, the interrupt request is not generated, allowing masking facility for the interrupt.

Mode 1 : Port A output operation.

Fig. 12.11 (a) shows Port A configured as an output port (when in Mode 1) along with the control word and control signals (for handshaking with a peripheral). When the control

word (as in Fig. 12.11 (a)) is loaded into the control register, Group A is configured in Mode 1 with Port A as an output port. The CPU can send data to a peripheral (like a display device) through Port A of the 8255.

The $\overline{\text{OBF}}_A$ output (Output Buffer Full) goes low on the rising edge of the $\overline{\text{WR}}$ signal (when the CPU writes data into the 8255). The $\overline{\text{OBF}}_A$ output from 8255 can be used as a strobe input to the peripheral to latch the contents of Port A. The peripheral responds to

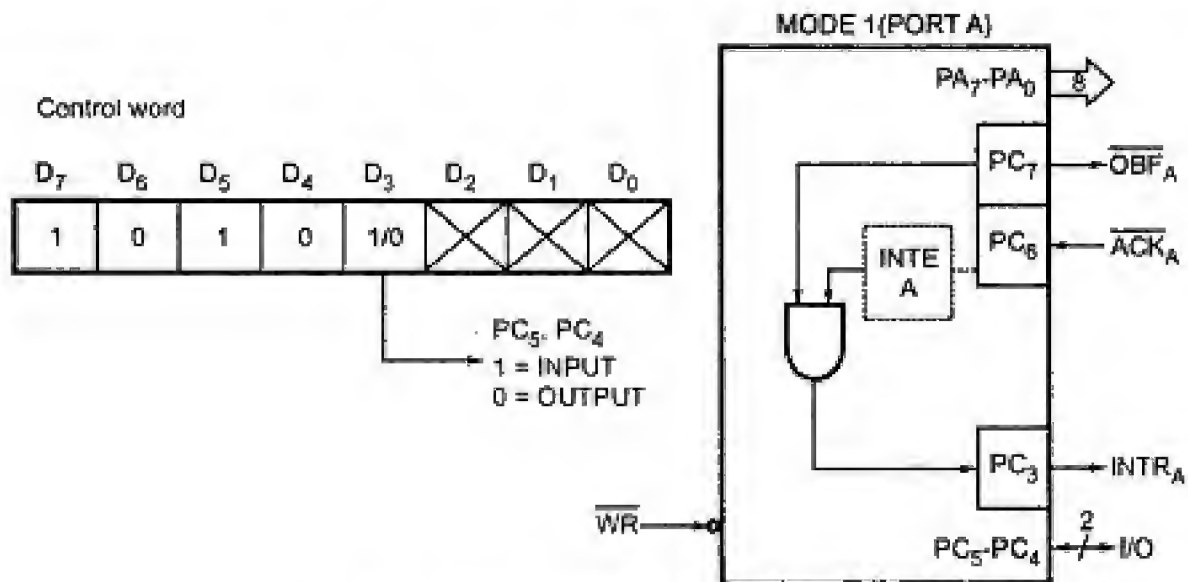


Fig. 12.11 (a) Port A in mode 1

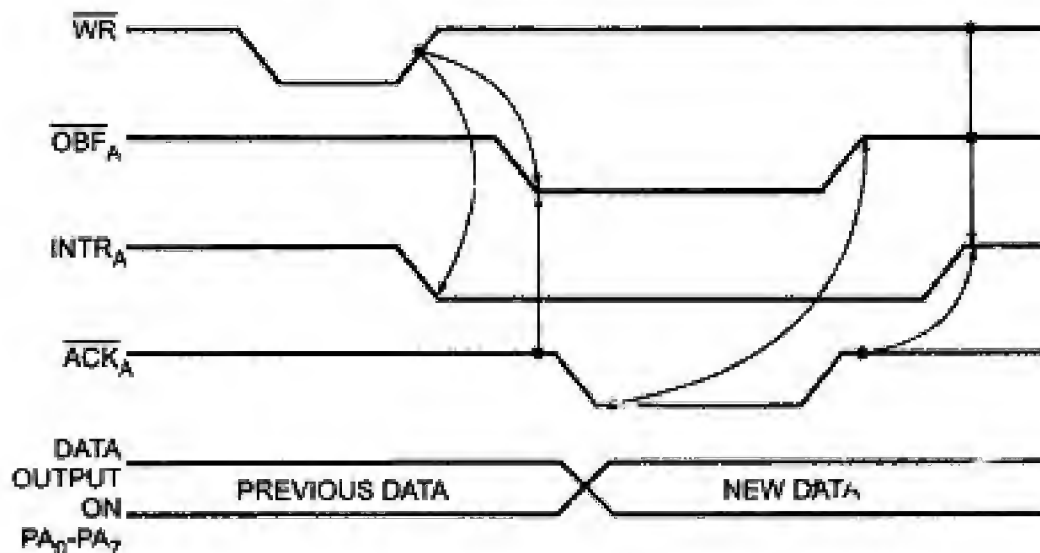


Fig. 12.11 (b) Port A in mode 1 (Output)

the receipt of data by making the \overline{ACK}_A input of the 8255 low, thus acknowledging that it has received the data sent by the CPU through Port A. The \overline{ACK}_A low sets the OBF_A signal, which can be polled by the CPU through \overline{OBF}_A of the status word to load the next data when it is high again.

$INTR_A$ is an active high output of the 8255 which is made high (if the associated $INTE_A$ flip-flop is set) when \overline{ACK}_A is made high again by the peripheral, and when \overline{OBF}_A goes high again (see timing diagram in Fig. 12.11(b)). It can be used to interrupt the CPU whenever the output buffer is empty. It is reset by the falling edge of \overline{WR} when the CPU writes data onto Port A. It can be enabled or disabled by writing a '1' or a '0' respectively to PC_0 in the BSR mode.

Mode 1 : Port B output operation.

Fig. 12.12 shows Port B as an output port when in Mode 1. The operation of Port B is similar to that of Port A. $INTR_A$ is controlled by writing a '1' or a '0' to PC_2 in the BSR mode. The status word is accessed by issuing a Read to Port C. The format of the status word when Ports A and B are Output ports in Mode 1 is shown in Fig. 12.13.

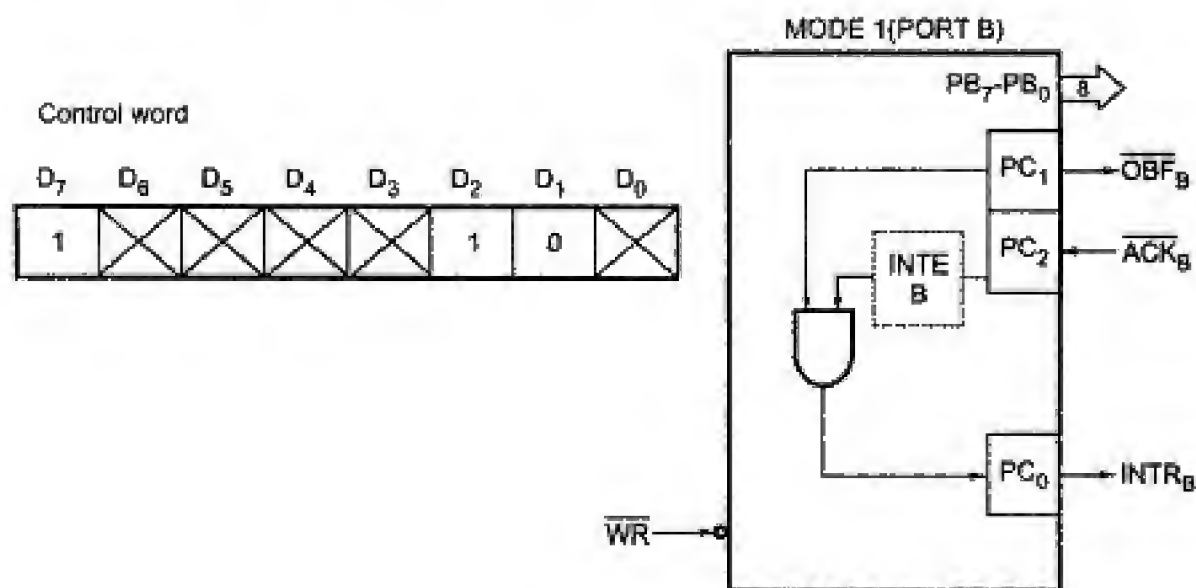


Fig. 12.12 Port B in mode 1 (Output)

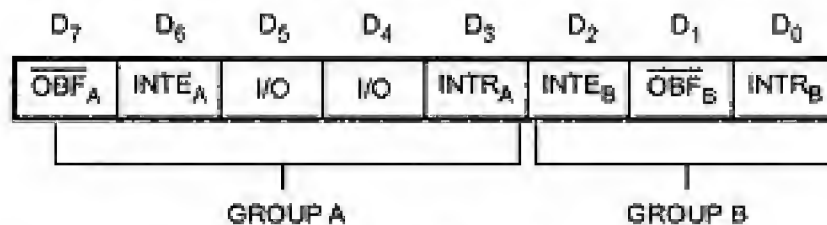


Fig. 12.13 Mode 1 status word (Output)

12.6.3 Programming in Mode 2 (Strobes Bi-directional Bus I/O)

When the 8255 is operated in Mode 2 (by loading the appropriate control word), Port A can be used as a bi-directional 8-bit I/O bus using for handshaking. Port B can be programmed in Mode 0 or in Mode 1. When Port B is programmed in mode 1, PC₀ - PC₂ lines of Port C are used as handshaking signals.

Fig. 12.14 shows the control word that should be loaded into the control port to configure 8255 in Mode 2.

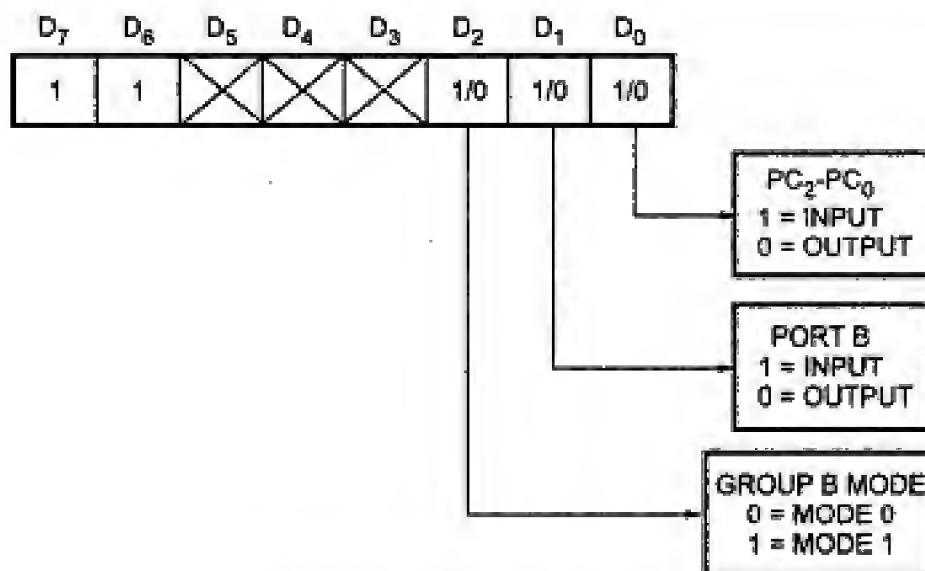


Fig. 12.14 Mode 2 control word

Mode 2 : Control signals

INTR (Interrupt Request) : A 'high' on this output can be used to interrupt the CPU for input or output operations.

Output Control Signals :

$\overline{\text{OBF}}_A$ (Output Buffer Full)

This is an active low output which indicates that the CPU has written data into Port A.

$\overline{\text{ACK}}_A$ (Acknowledge)

This is an active low input signal (generated by the peripheral) which enables the tri-state output buffer of Port A and makes Port A data available to the peripheral. In Mode 2, Port A outputs are in tri-state until enabled.

INTE 1

This is the flip-flop associated with Output Buffer Full. INTE 1 can be used to enable or disable the interrupt by setting or resetting PC_6 in the BSR Mode.

Input Control Signals : **$\overline{\text{STB}}$ (Strobe Input)**

This is an active low input signal which enables Port A to latch the data available at its input.

IBF (Input Buffer Full Flip-Flop)

This is an active high output which indicates that data has been loaded into the input latch of Port A.

INTE 2

This is an Interrupt enable flip-flop associated with Input Buffer Full. It can be controlled by setting or resetting PC_4 in the BSR Mode.

Mode 2 : Port A operation.

Fig. 12.15 shows Port A and associated control signals when 8255 is in Mode 2. Interrupts are generated for both output and input operations on the same INTR_A (PC_3) line.

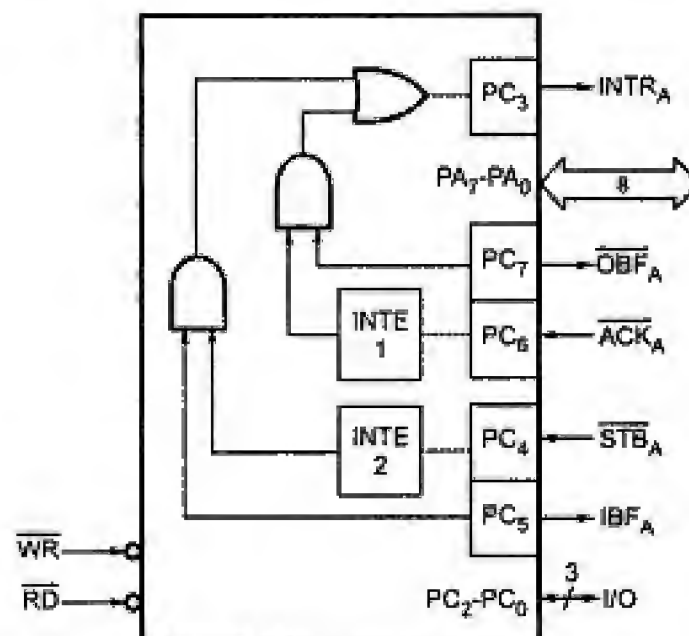


Fig. 12.15 Mode 2 operation

Status Word in Mode 2

The status word for Mode 2 (accessed by reading Port C) is shown in Fig. 12.17. D_7 - D_3 of the status word carry information about \overline{OBF}_A , $INTE_1$, IBF_A , $INTE_2$, $INTR_A$. The status of the bits D_2 - D_0 depends on the mode setting of Group B. If B is programmed in Mode 0, D_2 - D_0 are the same as PC_2 - PC_0 (simple I/O); however if B is in Mode 1, D_2 - D_0 carry information about the control signals for Port B (as in Fig. 12.10, or Fig. 12.13), depending upon whether Port B is an Input port or Output port respectively.

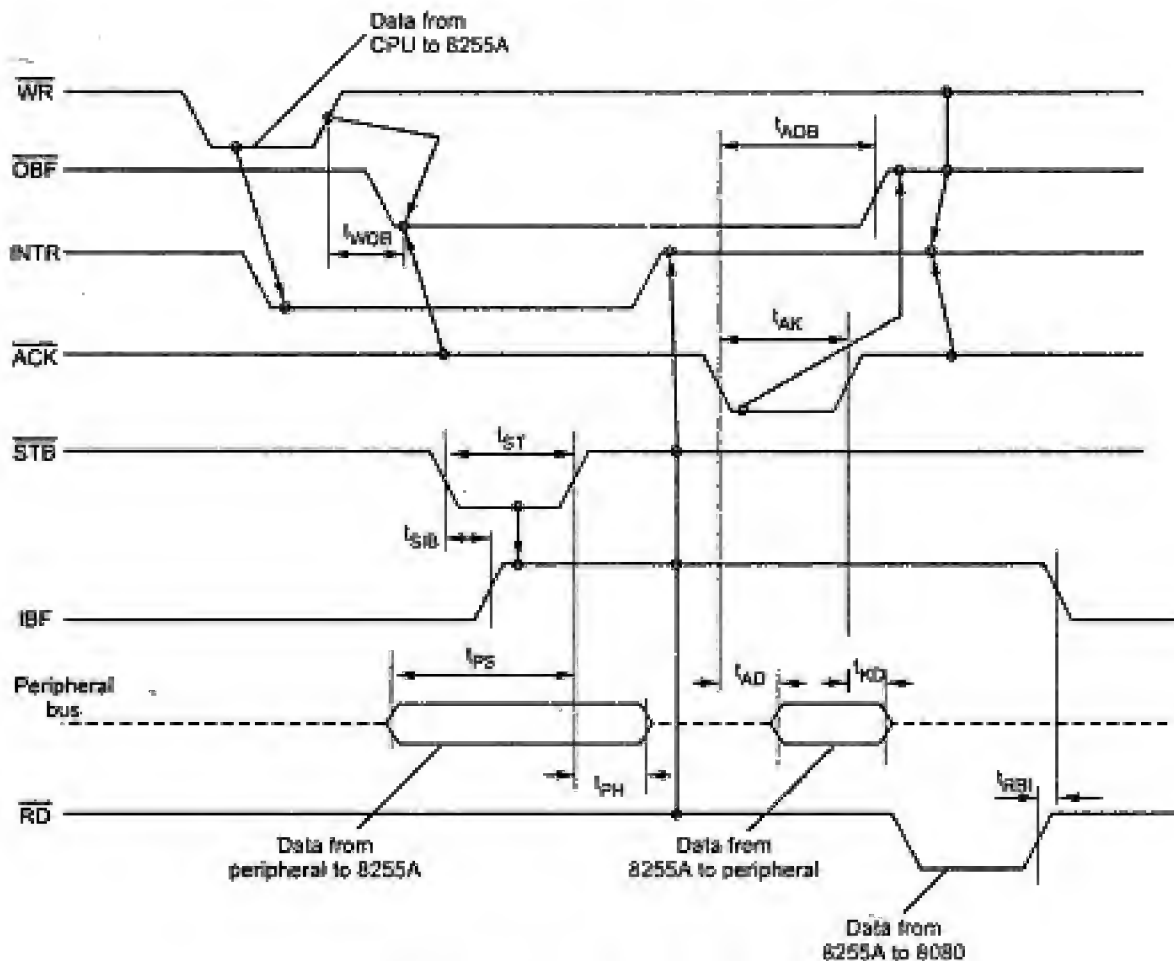


Fig. 12.16 Mode 2 waveform

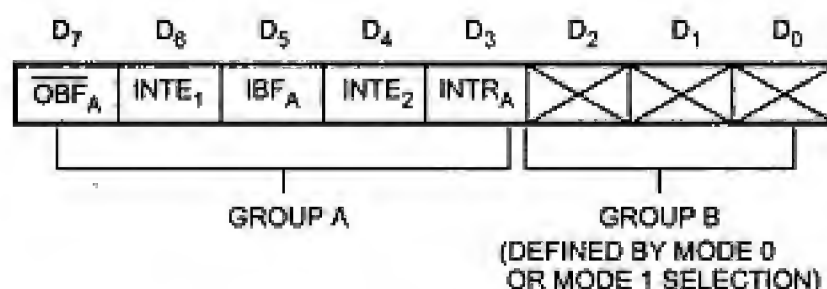


Fig. 12.17 Status word for mode 2

Mode Definition Summary

| | MODE 0 | | | MODE 1 | | | MODE 2 | |
|--|--|--|--|--|--|--|--|--|
| | IN | OUT | | IN | OUT | | GROUP A ONLY | |
| | PA ₀ PA ₁ PA ₂ PA ₃ PA ₄ PA ₅ PA ₆ PA ₇ | IN IN IN IN IN IN IN IN | | OUT OUT OUT OUT OUT OUT OUT OUT | IN IN IN IN IN IN IN IN | | OUT OUT OUT OUT OUT OUT OUT OUT | ↔ ↔ ↔ ↔ ↔ ↔ ↔ ↔ |
| | MODE 0 | | | MODE 1 | | | MODE 2 | |
| | IN | OUT | | IN | OUT | | GROUP A ONLY | |
| | PB ₀ PB ₁ PB ₂ PB ₃ PB ₄ PB ₅ PB ₆ PB ₇ | IN IN IN IN IN IN IN IN | | OUT OUT OUT OUT OUT OUT OUT OUT | IN IN IN IN IN IN IN IN | | OUT OUT OUT OUT OUT OUT OUT OUT | - - - - - - - - |
| | MODE 0 | | | MODE 1 | | | MODE 2 | |
| | IN | OUT | | IN | OUT | | GROUP A ONLY | |
| | PC ₀ PC ₁ PC ₂ PC ₃ PC ₄ PC ₅ PC ₆ PC ₇ | IN IN IN IN IN IN IN IN | | OUT OUT OUT OUT OUT OUT OUT OUT | INTR _B IBF _B STB _B INTR _A STB _A IBF _A I/O I/O | | INTR _B OBF _B ACK _B INTR _A I/O I/O ACK _A OBF _A | I/O I/O I/O INTR _A STB _A IBF _A ACK _A OBF _A |

Mode 0
or
Mode 1
Only

12.7 Interfacing 8255 in I/O Mapped I/O

Fig. 12.18 shows the interfacing of 8255 with 8085 in I/O mapped I/O technique. Here \overline{RD} and \overline{WR} signals are activated when $\overline{IO/\overline{M}}$ signal is high, indicating I/O bus cycle. Reset out signal from 8085 is connected to the RESET signal of the 8255.

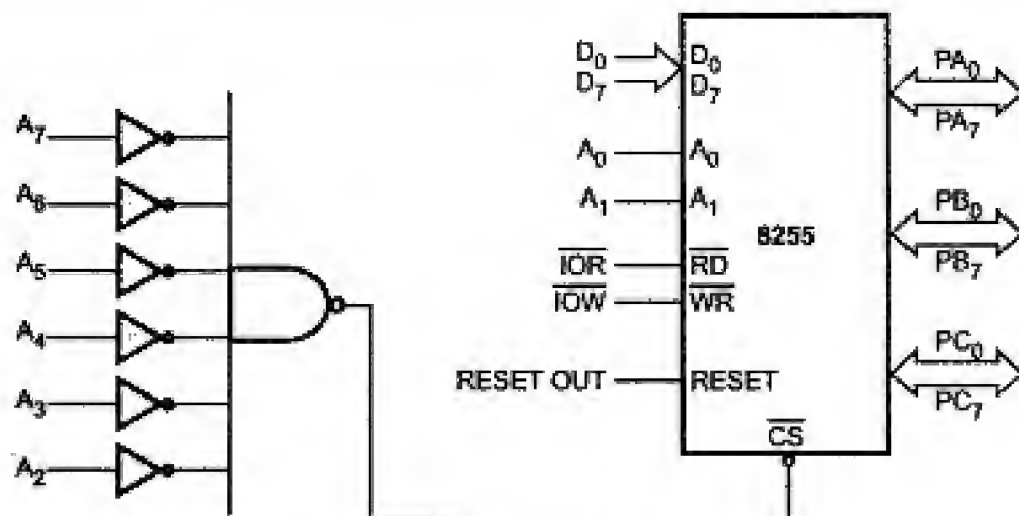


Fig. 12.18 Interfacing of 8255 in I/O mapped I/O

I/O Map :

| Ports / Control Register | Address Lines | | | | | | | | Address |
|--------------------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|---------|
| | A ₇ | A ₆ | A ₅ | A ₄ | A ₃ | A ₂ | A ₁ | A ₀ | |
| Port A | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 00H |
| Port B | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 01H |
| Port C | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 02H |
| Control Register | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 03H |

12.8 Interfacing 8255 in Memory Mapped I/O

Fig. 12.19 shows the interfacing of 8255 with 8085 in memory mapped I/O technique. Here \overline{RD} and \overline{WR} signals are activated when $\overline{IO/\overline{M}}$ signal is low, indicating memory bus cycle. To get absolute address, all remaining address lines ($A_{15} - A_2$) are used to decode the address for 8255. Other signal connections are same as in I/O mapped I/O.

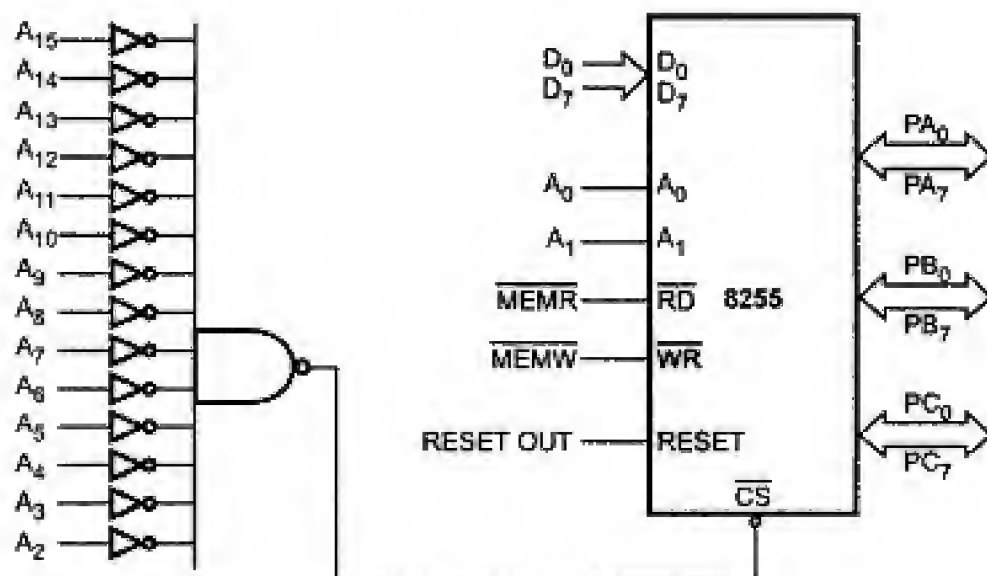


Fig. 12.19 Interfacing 8255 in memory mapped I/O

I/O Map :

| Ports/Control | Address Lines | | | | | | | | | | | | | | | | Address |
|------------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|---------|
| Register | A ₁₅ | A ₁₄ | A ₁₃ | A ₁₂ | A ₁₁ | A ₁₀ | A ₉ | A ₈ | A ₇ | A ₆ | A ₅ | A ₄ | A ₃ | A ₂ | A ₁ | A ₀ | |
| Port A | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0000H |
| Port B | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0001H |
| Port C | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0002H |
| Control Register | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0003H |

Lab Experiment 63 : Hardware and software for flashing LEDs.

Statement : Design a system that will cause 4 LEDs to flash 10 times when a push button switch is pressed. Use 8255. Assume persistence of vision to be 0.1 seconds.

Hardware : Fig. 12.20 shows the interfacing scheme. In this scheme the LEDs are connected to port C in sinking mode. Thus when port C pin is low LED is ON and when Port C pin is high LED is OFF. Port A pin 0 is used to read status of push button switch.

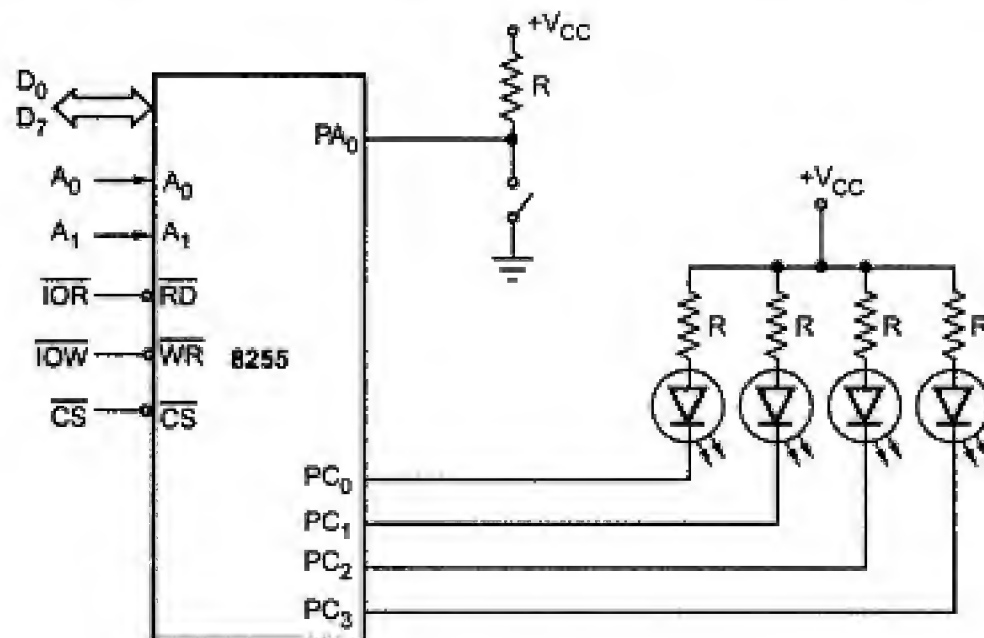


Fig. 12.20

Software :**Program**

```

                LXI SP, 2000H      ; Initialize stack pointer
                MVI A, 90 H        ;
                OUT CR              ; Initialize 8255
BACK :          IN PA              ; [Read status
                ANI 01             ; of push
                JNZ BACK           ; button]
                MVI B, 0AH         ; Initialize counter
AGAIN :         MVI A, 00H         ; Load data to light LEDs
                OUT PC             ; Send data on port C
                CALL Delay         ; Call. Delay of 0.1 sec
                MVI A FFH         ; Load data to switch off LEDs
                OUT PC             ; Send data on port C
                CALL Delay         ; Call Delay of 0.1 sec
                DCR B              ; Decrement count
                JNZ AGAIN          ; If not zero repeat
                JMP BACK           ; Jump back to read status

```

Review Questions

1. Discuss the organization and architecture of 8255 programmable peripheral interface IC with a functional block diagram.
2. Illustrate the different modes of operation of 8255.
3. Explain the control word formation with suitable examples.
4. Draw the interfacing scheme of 8255 and 8085 in I/O mapped I/O mode.
5. Draw the interfacing scheme of 8255 and 8085 in memory mapped I/O mode.
6. Explain the bit set/reset mode of 8255.
7. Write a program in assembly language using 8255 to generate a square wave of 1 MHz frequency.
8. An 8255 is used with Port A as output, Port B as input and with Port C used for handshaking for Port A and Port B. The 8255 is enabled when A_{15} to A_8 is 001101XX.
 - i) Write the instruction sequence to program the 8255 for this mode of operation.
 - ii) Write the format of the status word obtained if PORT C is read.
 - iii) Draw the scheme of connection required.



Interfacing to External World

13.1 Introduction

Any application of a microcontroller based system requires the transfer of data between external circuitry to the microcontroller and microcontroller to the external circuitry. User can give information to the microcontroller using keyboard and user can see the result or output information from the microcontroller with the help of display device. The transfer of data between keyboard and microcontroller, and microcontroller and display device is called input/output data transfer or **I/O data transfer**.

We have seen that 8051 has internal data and code memory with limited memory capacity. This memory capacity may not be sufficient for some applications. In such situations, we have to connect external ROM/EPROM and RAM to 8051 microcontroller to increase the memory capacity.

In this chapter, we are going to study the accessing of data from external world and code memory and interfacing of input and output devices to the 8051 microcontroller.

13.2 External RAM and ROM

We know that ROM is used as a program memory and RAM is used as a data memory. Let us see how 8051 accesses these memories.

13.2.1 Program Memory

Fig. 13.1 shows a map of the 8051 program memory.

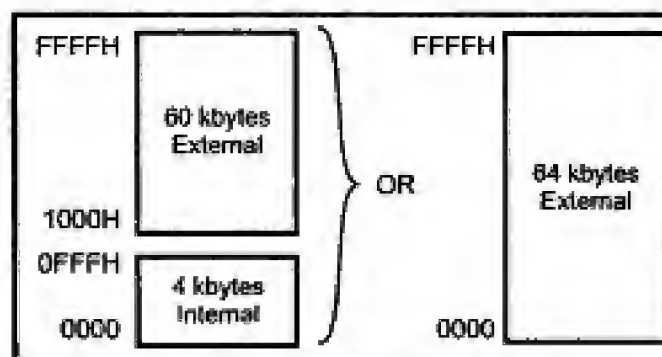


Fig. 13.1 The 8051 program memory

In 8051, when the \overline{EA} pin is connected to V_{CC} , program fetches to addresses 0000H through 0FFFH are directed to the internal ROM and program fetches to addresses 1000H through FFFFH are directed to external ROM/EPROM. On the other hand when \overline{EA} pin is grounded, all addresses (0000H to FFFFH) fetched by program are directed to the external ROM/EPROM. The \overline{PSEN} signal is used to activate output enable signal of the external ROM/EPROM, as shown in the Fig. 13.2.

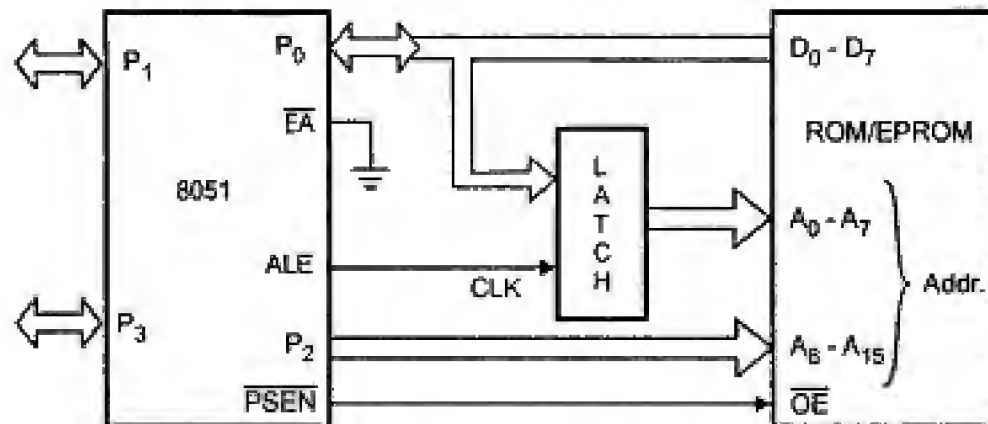


Fig. 13.2 Accessing external program memory

As shown in the Fig. 13.2, the port 0 is used as a multiplexed address/bus. It gives lower order 8-bit address in the initial T-cycle and later it is used as a data bus. The 8-bit address is latched using external latch and ALE signal generated by 8051. The port 2 provides the higher order 8-bit address. Fig. 13.3 shows the timing waveforms for external program memory read cycle.

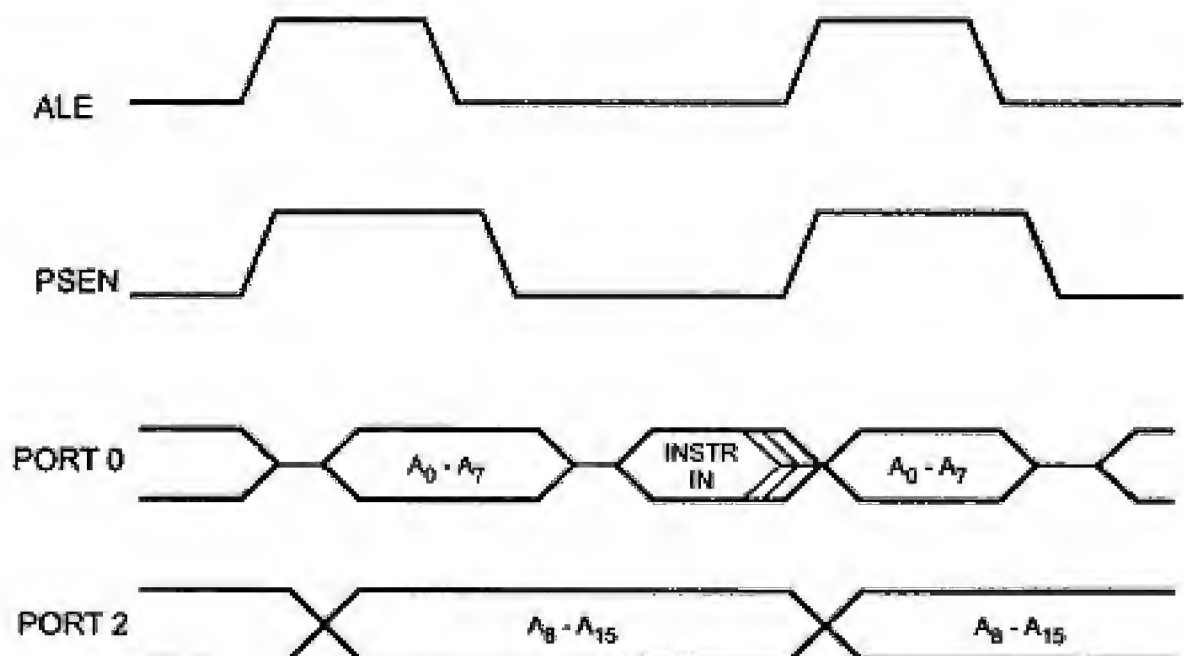


Fig. 13.3 Timing waveforms for external program memory read cycle

The lower part of program memory stores the vector addresses for various interrupt service routines. Fig. 13.4 shows the vector address map. Each interrupt is assigned with a fixed location in program memory. For example, external interrupt 0 is assigned to location 0003H. The interrupt service locations are spaced at 8-byte intervals such as 0003H for External Interrupt 0, 000BH for Timer 0, 0013H for External Interrupt 1, 001BH for Timer 1, etc. If interrupt is going to be used, its service routine must begin at corresponding location. If the interrupt is not going to be used, its service location is available as general purpose program memory.

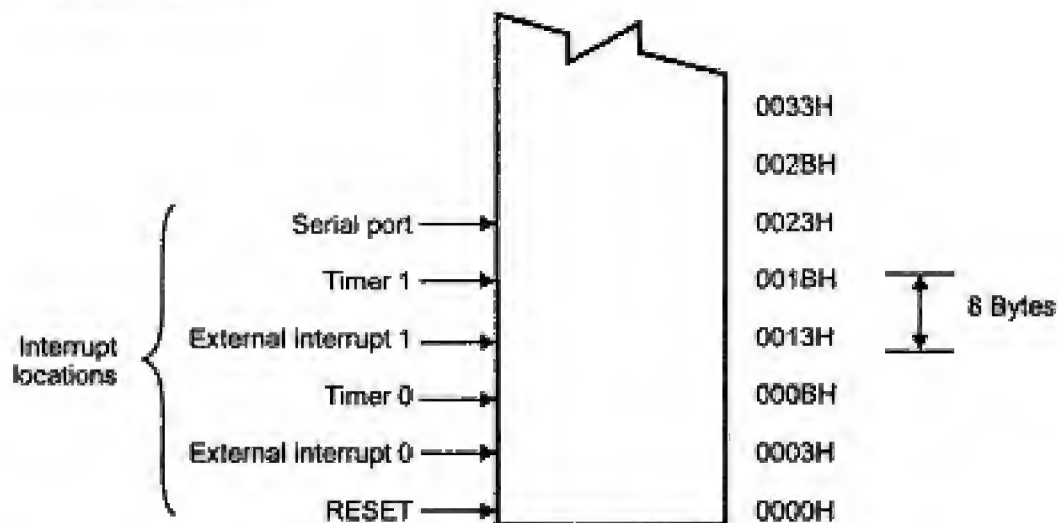


Fig. 13.4 Interrupt/Vector locations in the lower part of program memory

Instructions to Access External ROM / Program Memory

The Table 13.1 explains the instructions to access external ROM/program memory.

| Mnemonic | Operation |
|--------------------|--|
| MOVC A, @ A + DPTR | Copy the contents of the external ROM address formed by adding A and the DPTR, to A. |
| MOVC A, @ A + PC | Copy the contents of the external ROM address formed by adding A and the PC, to A. |

Table 13.1

13.2.2 Data Memory

Fig. 13.5 shows a map of the 8051 data memory

The 8051 can address upto 64 kbytes of external data memory. The "MOVX" instruction is used to access the external data memory. The internal data memory space for 8051 is divided into three blocks : Lower 128 bytes, Upper 128 bytes and SFRs. The upper addresses and SFRs occupy the same block of address space, 80H through FFH, although they are physically separate entities. As shown in the Fig. 13.5, the upper address space is

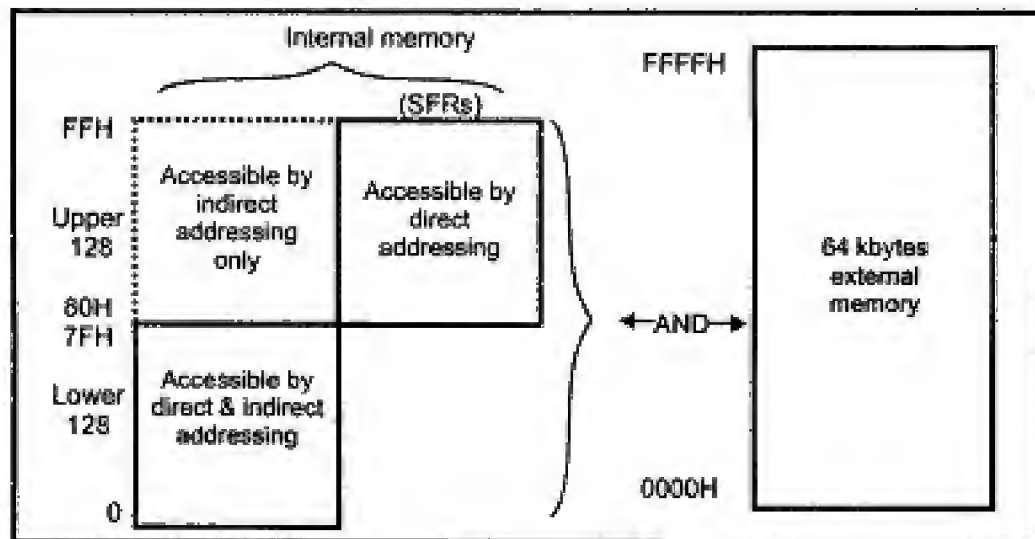


Fig. 13.5 A map of the 8051 data memory

accessible by indirect addressing only and SFRs are accessible by direct addressing only. On the other hand, lower address space can be accessed either by direct addressing or by indirect addressing.

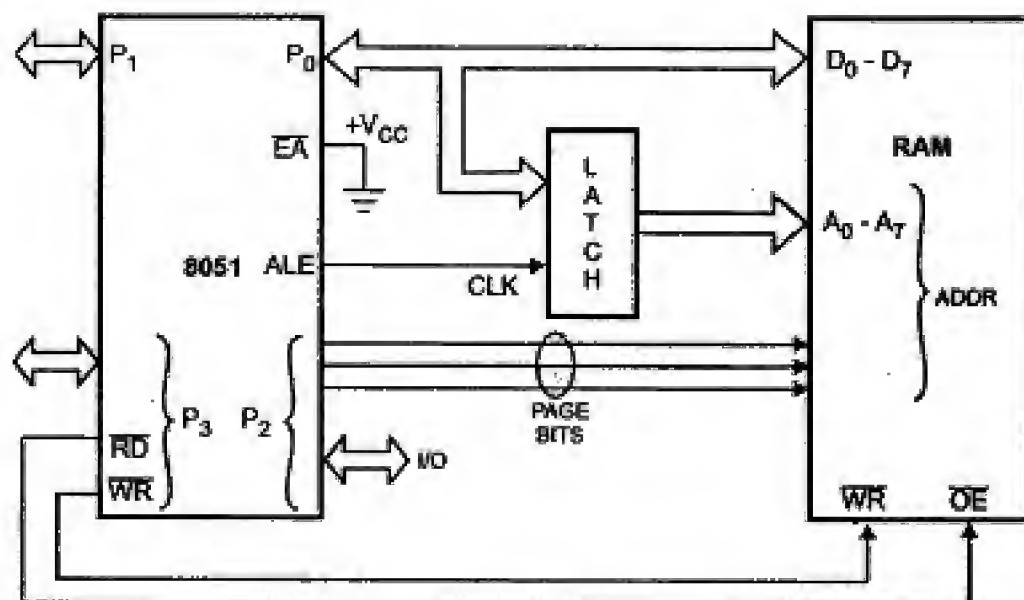


Fig. 13.6 Accessing external data memory

Fig. 13.6 shows the circuit diagram for connecting external data memory. The multiplexed address/data bus provided by port 0 is demultiplexed by external latch and ALE signal. Port 2 gives the higher order address bus. The \overline{RD} and \overline{WR} signals from 8051 selects the memory read and memory write operation, respectively.

Fig. 13.7 (a) and (b) show the timing waveforms for external data memory read and write cycles, respectively.

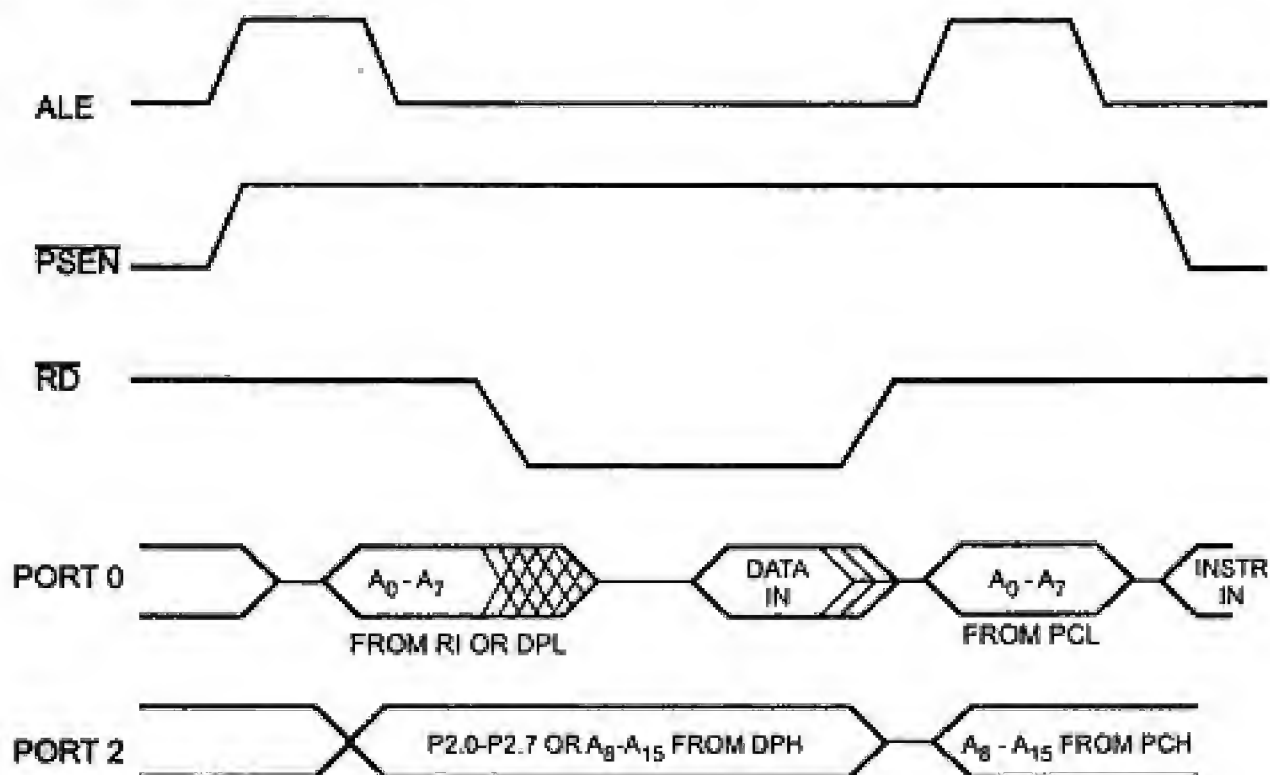


Fig. 13.7 (a) Timing waveforms for external data memory read cycle

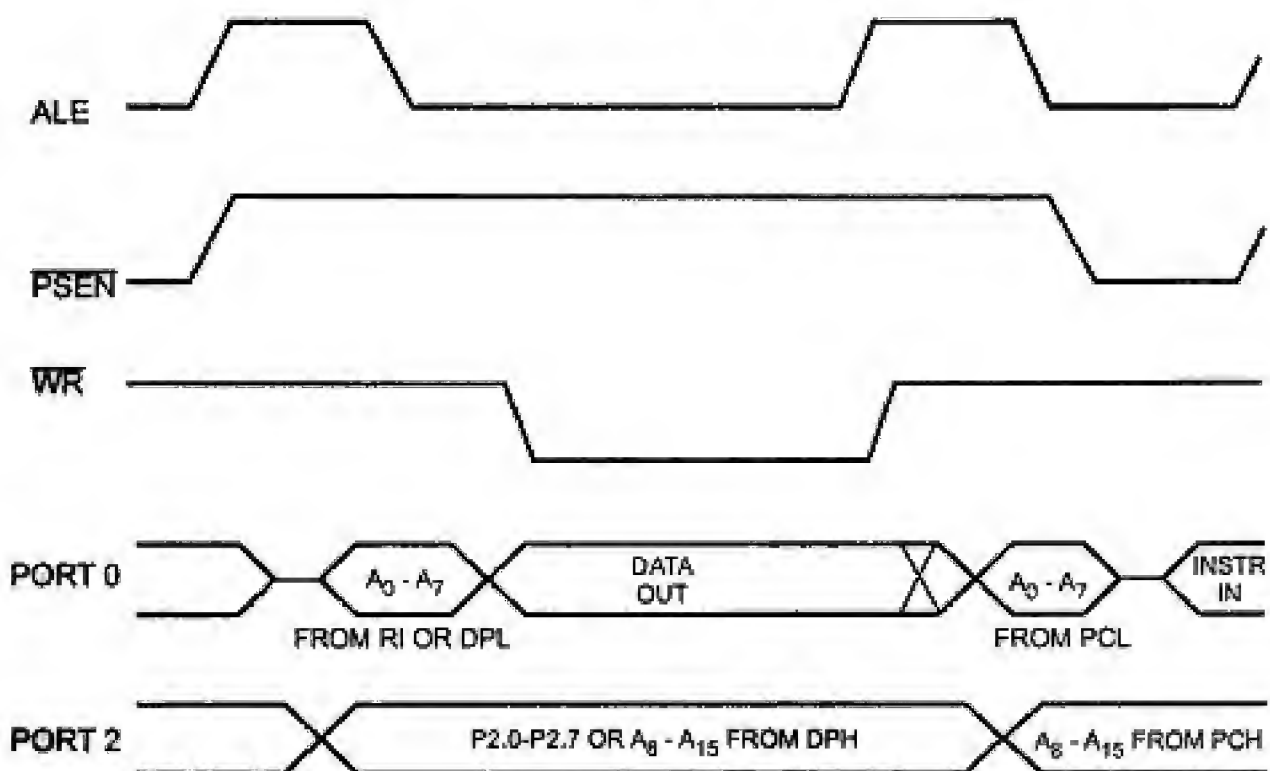


Fig. 13.7 (b) Timing waveforms for external data memory write cycle

Instructions to Access External Data Memory

The Table 13.2 explains the instruction to access external data memory.

| Mnemonic | Operation |
|---------------|---|
| MOVX A, @Rp | Copy the contents of the external address in Rp to A. |
| MOVX A, @DPTR | Copy the contents of the external address in DPTR to A. |
| MOVX @Rp, A | Copy data from A to the external address in Rp. |
| MOVX @DPTR, A | Copy data from A to the external address in DPTR. |

Table 13.2

13.2.3 Important Points to Remember in Accessing External Memory

- All external data moves with external ROM or external RAM involve the A register.
- While accessing external memory, R_p can address 256 bytes and DPTR can address 64 kbytes.
- MOVX instruction is used to access external RAM or I/O addresses.
- When PC is used to access external ROM, it is incremented by 1 (to point to the next instruction) before it is added to A to form the physical address of external ROM.

13.2.4 Memory Interfacing

We know that read/write memories consist of an array of registers, in which each register has unique address. The size of the memory is $N \times M$ as shown in Fig. 13.8 (a) where N is the number of registers and M is the word length, in number of bits.

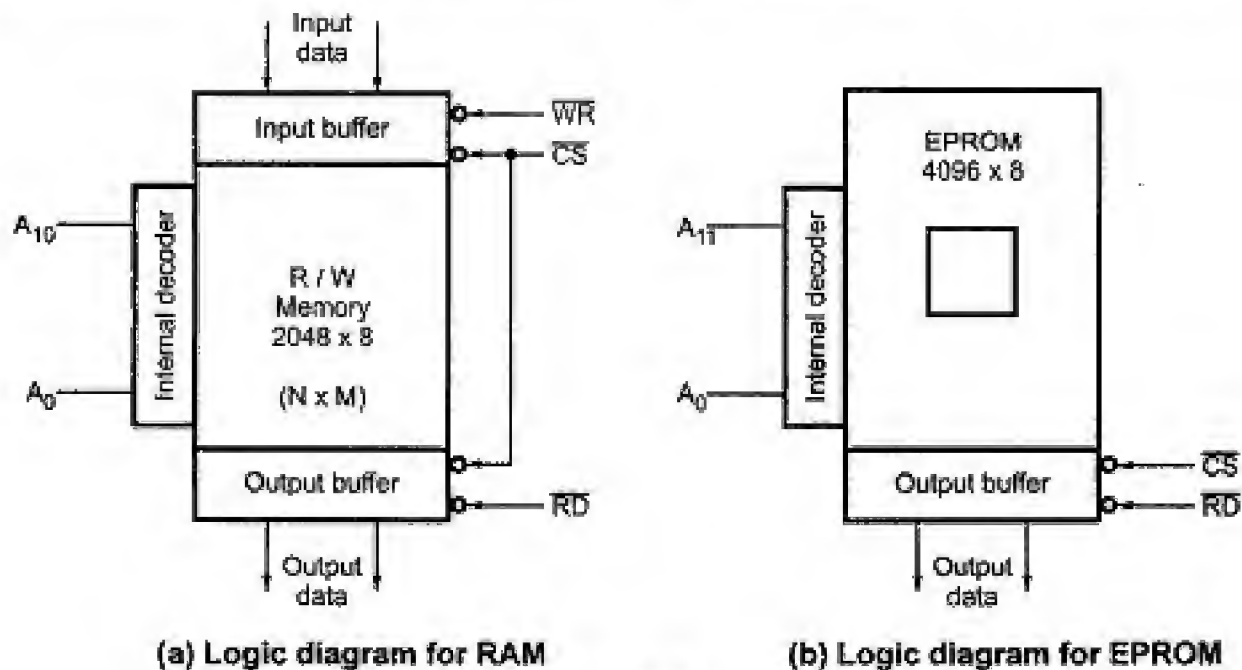


Fig. 13.8

For Example : If memory is having 12 address lines and 8 data lines, then
 Number of registers/memory locations = $2^N = 2^{12} = 4096$.
 Word length = Mbit = 8-bit .

For Example : If memory has 8192 memory locations, then it has 13 address lines.

The Table 13.3 summarizes the memory capacity and address lines required for memory interfacing.

| Memory Capacity | Address Lines Required |
|-------------------------------|------------------------|
| 1 K = 1024 memory locations | 10 |
| 2 K = 2048 memory locations | 11 |
| 4 K = 4096 memory locations | 12 |
| 8 K = 8192 memory locations | 13 |
| 16 K = 16384 memory locations | 14 |
| 32 K = 32768 memory locations | 15 |
| 64 K = 65536 memory locations | 16 |

Table 13.3

As shown in the Fig. 13.8 (a) memory chip has 11 address lines A_{10} - A_0 , one chip select (\overline{CS}), and two control lines. Read (\overline{RD}) to enable output buffer and write (\overline{WR}) to enable the input buffer. The internal decoder is used to decode the address lines. Fig. 13.8 (b) shows the logic diagram of a typical EPROM (Erasable Programmable Read Only Memory)

with 4096 (4 K) registers. It has 12 address lines A_{11} - A_0 , one chip select (\overline{CS}), one read control signal. Since EPROM is a read only memory, it does not require the (\overline{WR}) signal.

The memory interfacing requires to :

- Select the chip.
- Identify the register.
- Enable the appropriate buffer.

Microprocessor/microcontroller system includes memory devices and I/O devices. It is important to note that microprocessor can communicate (read/write) with only one device at a time, since the data, address and control buses are common for all the devices. In order to communicate with memory or I/O devices, it is necessary to decode the address from the microprocessor/microcontroller. The following section describes common address decoding techniques.

Address Decoding Techniques :

- Absolute decoding/Full decoding
- Linear decoding/Partial decoding

Absolute decoding

In absolute decoding technique, all the higher address lines are decoded to select the memory chip, and the memory chip is selected only for the specified logic levels on these high-order address lines; no other logic levels can select the chip. Fig. 13.9 shows the memory interface with absolute decoding. This addressing technique is normally used in large memory systems.

Memory Map

| Memory ICs | A_{15} | A_{14} | A_{13} | A_{12} | A_{11} | A_{10} | A_9 | A_8 | A_7 | A_6 | A_5 | A_4 | A_3 | A_2 | A_1 | A_0 | Address |
|---------------------------|----------|----------|----------|----------|----------|----------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|---------|
| Starting address of EPROM | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0000H |
| End address of EPROM | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 03FFH |
| Starting address of RAM | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2000H |
| End address of RAM | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 23FFH |

Table 13.4

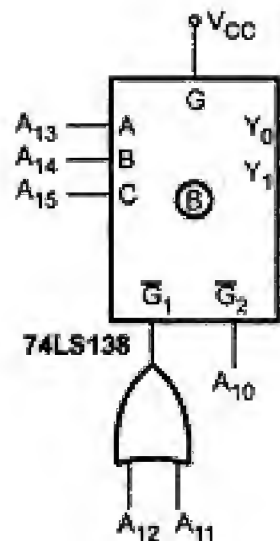


Fig. 13.9 Absolute decoding technique

Linear decoding

In small systems, hardware for the decoding logic can be eliminated by using individual high-order address lines to select memory chips. This is referred to as linear decoding. Fig. 13.10 shows the addressing of RAM with linear decoding technique. This technique is also called **partial decoding**. It reduces the cost of decoding circuit, but it has a drawback of multiple addresses (shadow addresses).

Fig. 13.10 shows the addressing of RAM with linear decoding technique. A_{15} address line, is directly connected to the chip select signal of EPROM and after inversion it is connected to the chip select signal of the RAM. Therefore, when the status of A_{15} line is 'zero', EPROM gets selected and when the status of A_{15} line is 'one' RAM gets selected. The status of the other address lines is not considered, since those address lines are not used for generation of chip select signals.

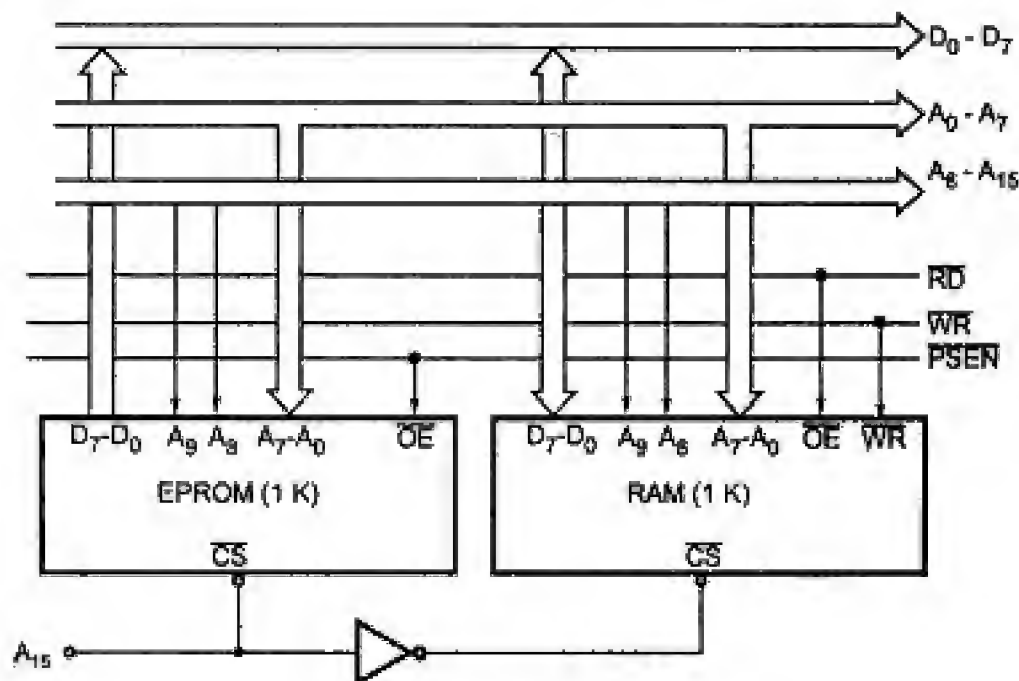


Fig. 13.10 Linear decoding

Memory Map :

| Memory ICs | A ₁₅ | A ₁₄ | A ₁₃ | A ₁₂ | A ₁₁ | A ₁₀ | A ₉ | A ₈ | A ₇ | A ₆ | A ₅ | A ₄ | A ₃ | A ₂ | A ₁ | A ₀ | Address |
|---------------------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|---------|
| Starting address of EPROM | 0 | X | X | X | X | X | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0000H |
| End address of EPROM | 0 | X | X | X | X | X | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 03FFH |
| Starting address of RAM | 1 | X | X | X | X | X | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 8000H |
| End address of RAM | 1 | X | X | X | X | X | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 83FFH |

Table 13.5

13.2.5 Interfacing Example

➡ **Example 13.1 :** Give the complete block schematic of an 8051 based system having following specifications.

64 kB of program memory.

64 kB of data memory.

Make use of 16 K × 8 bit memory chips and 74 LS 138 decoders.

Indicate clearly, the addresses selected for the memory chips.

Solution :

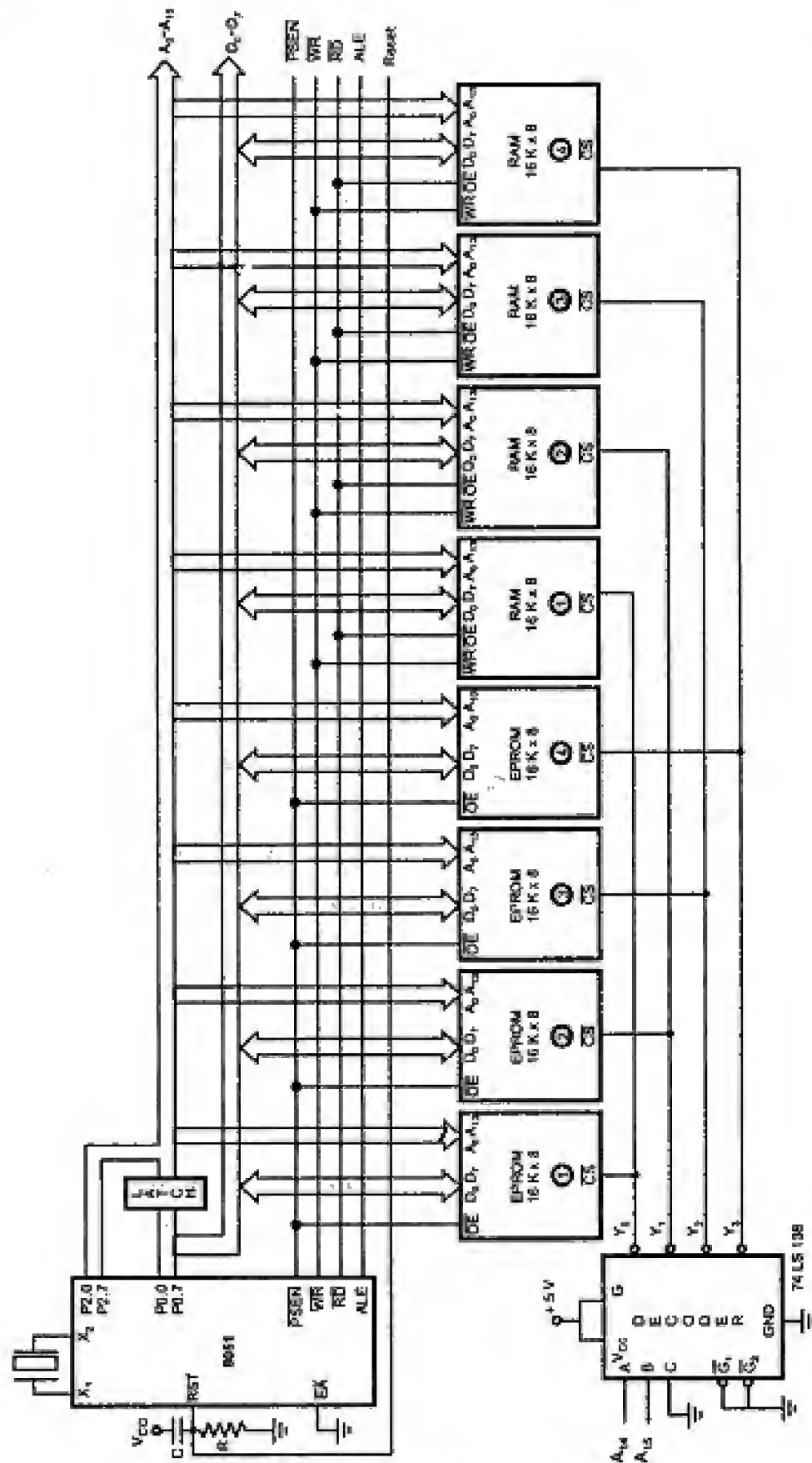


Fig. 13.11

Memory Map :

| Memory | A ₁₅ | A ₁₄ | A ₁₃ | A ₁₂ | A ₁₁ | A ₁₀ | A ₉ | A ₈ | A ₇ | A ₆ | A ₅ | A ₄ | A ₃ | A ₂ | A ₁ | A ₀ | Address |
|--------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|---------|
| EPROM1 Start | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0000H |
| EPROM1 End | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 3FFFH |
| EPROM2 Start | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4000H |
| EPROM2 End | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 7FFFH |
| EPROM3 Start | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 8000H |
| EPROM3 End | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | BFFFH |
| EPROM4 Start | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | C000H |
| EPROM4 End | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | FFFFH |
| RAM1 Start | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0000H |
| RAM1 End | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 3FFFH |
| RAM2 Start | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4000H |
| RAM2 End | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 7FFFH |
| RAM3 Start | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 8000H |
| RAM3 End | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | BFFFH |
| RAM4 Start | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | C000H |
| RAM4 End | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | FFFFH |

Table 13.6**13.3 8051 I/O Expansion using 8255**

As seen earlier, for interfacing external memory to the 8051, port 0 and port 2 are used as multiplexed address/data bus and a higher order address bus respectively. If the circuit needs the on chip peripherals (e.g. serial I/O and Interrupts) then only 1 port is available for I/O. In such situations, I/O expansion is necessary and it is achieved by using 8255. The Fig. 13.12 shows the expanded I/O ports using 8255. Data bus of 8255 is connected to the port 0. Address lines A₀ and A₁, after latches are connected to A₀ and A₁ of the 8255.

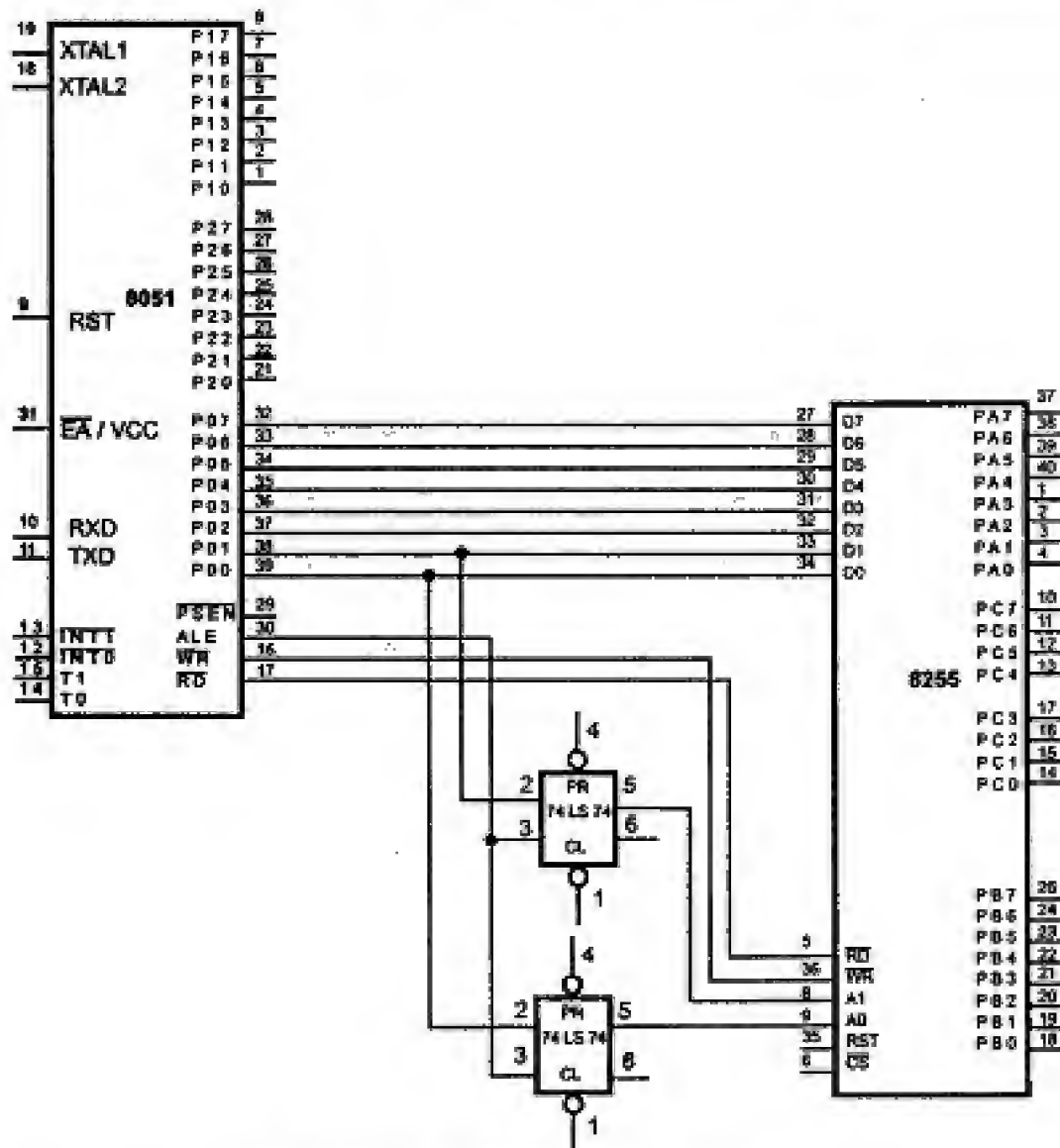


Fig. 13.12 I/O expansion using 8255

13.4 Interfacing Keyboard

For interfacing keyboard to the microprocessor/microcontroller based systems, usually push button keys are used. These push button keys when pressed, bounces a few times, closing and opening the contacts before providing a steady reading, as shown in the Fig. 13.13. Reading taken during bouncing period may be faulty. Therefore, microprocessor/microcontroller must wait until the key reach to a steady state; this is known as key debounce.

The problem of key bounce can be eliminated using key debounce technique, either hardware or software.

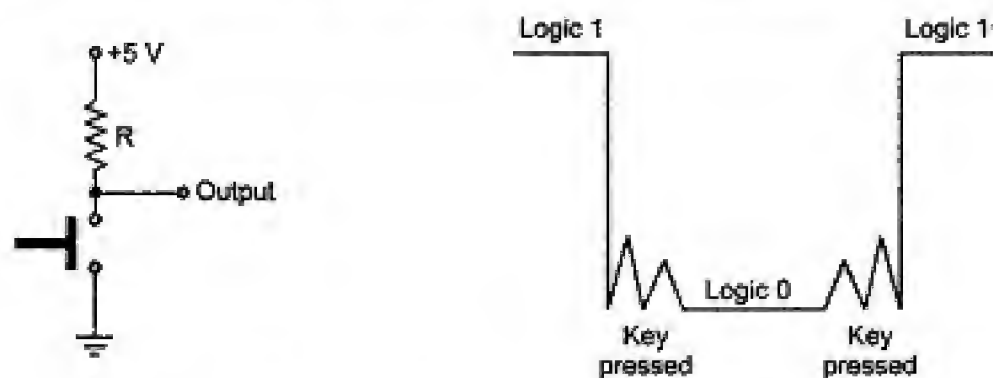


Fig. 13.13 Bouncing of key switch

13.4.1 Key Debounce using Hardware

| Key position | a | b | Y | c | d | \bar{Y} |
|-----------------|---|-----------|-----------|---|---|-----------|
| A | 0 | 0 | 1 | 1 | 1 | 0 |
| B | 1 | 1 | 0 | 0 | 0 | 1 |
| Between A and B | 1 | \bar{Y} | No change | Y | 1 | No change |

Table 13.7

Fig. 13.14 shows the circuit diagram of key debounce. It consists of flip-flop. The output of flip-flop shown in Fig. 13.14 is logic 1 when key is at position A (unpressed) and it is logic 0 when key is at position B, as shown in Table 13.7. It is important to note that, when key is in between A and B, output does not change, preventing bouncing of key output. In other words we can say that output does not change during transition period, eliminating key debouncing.

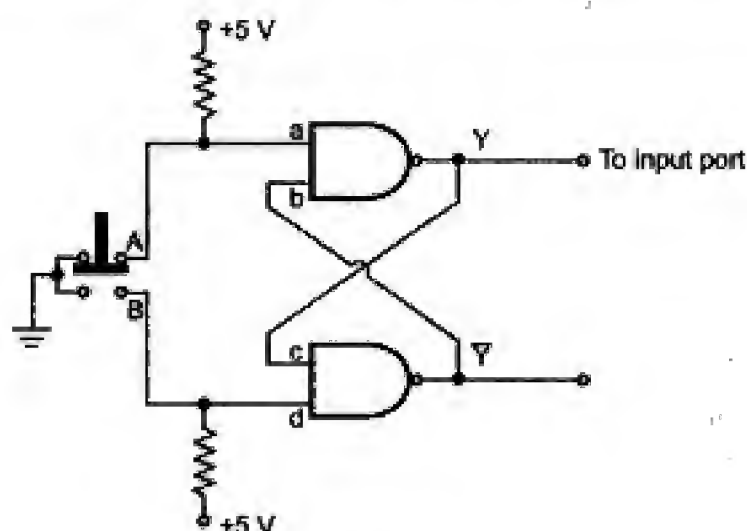


Fig. 13.14

13.4.2 Key Debouncing using Software

In the software technique, when a key press is found, the microprocessor/microcontroller waits for atleast 10 ms before it accepts the key as an input. This 10 ms period is sufficient to settle key at steady state. Fig. 13.15 shows the flowchart with key debounce technique.

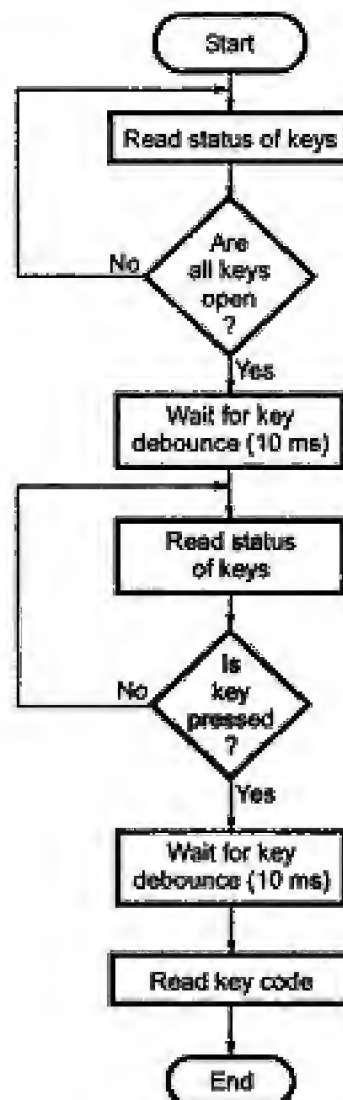


Fig. 13.15 Flowchart of key input with debounce

13.4.3 Simple Keyboard Interface

Fig. 13.16 shows simple keyboard interface.

Here eight keys are individually connected to specific pins of port P1. Each port pin gives the status of key connected to that pin. When port pin is logic 1, key is open, otherwise key is closed.

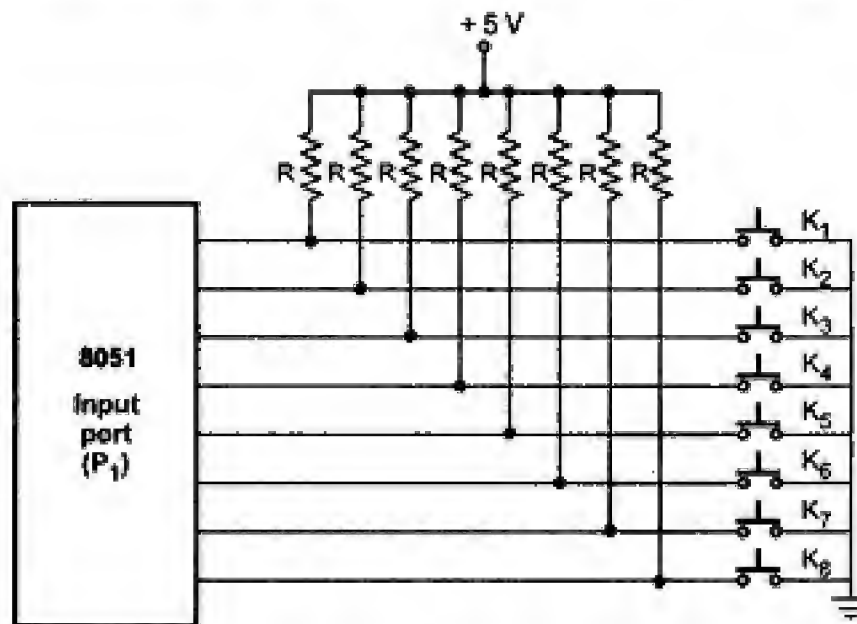


Fig. 13.16 Simple keyboard interface

Software routine to get keycode with key debounce.

```

START : MOV A, P1          ; Read key status
          ; check if keys are open
          CJNE A, #FFH, START ; if no, goto start otherwise
          ; continue
PRO :    LCALL DEBOUNCE_DELAY ; call debounce delay
AGAIN :  MOV A, P1          ; Read key status
          ; check if any key is pressed
          CJNE A, #FFH, PRO1  ; if no, goto AGAIN; otherwise
          ; continue
PRO1 :   LCALL DEBOUNCE_DELAY ; call debounce delay
          MOV A, P1          ; Get key code
          RET                ; Return from subroutine
  
```

This program reads status of all keys by getting data through P1 and compares it with FFH to check whether all keys are open. If all keys are open, instruction compare sets the zero flag, and the program waits for key debounce. After waiting about 10 ms, program checks the P1 for key press. If key press is found, program waits for another 10 ms as a key debounce period. After key debounce period, program reads the keycode from P1.

| Key | Keycode | | | | | | | |
|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
| | D ₇ | D ₆ | D ₅ | D ₄ | D ₃ | D ₂ | D ₁ | D ₀ |
| K ₁ | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| K ₂ | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| K ₃ | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |
| K ₄ | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |

| | | | | | | | | |
|----------------|---|---|---|---|---|---|---|---|
| K ₅ | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| K ₆ | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| K ₇ | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| K ₈ | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Table 13.8

13.4.4 Matrix Keyboard Interface

In simple keyboard interface one input line is required to interface one key and this number will increase with number of keys. Therefore, such technique is not suitable when it is necessary to interface large number of keys. To reduce number of connections keys are arranged in the matrix form as shown in the Fig. 13.17.

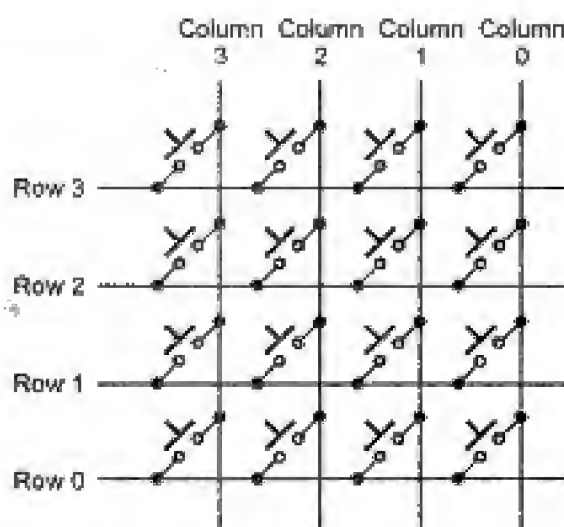


Fig. 13.17 Matrix keyboard

Fig. 13.17 shows sixteen keys arranged in four rows and four columns. When keys are open, row and column do not have any connection. When a key is pressed, it shorts corresponding one row and one column. This matrix keyboard requires eight lines to make all the connections instead of the sixteen lines required if the keys are connected individually, as shown in Fig. 13.16. Fig. 13.18 shows the interfacing of matrix keyboard. It requires two ports : an input port and an output port. Rows are connected to the input port referred to as returned lines, and columns are connected to the output port referred to as scan lines. We know that, when all keys are open, row and column do not have any connection. When any key is pressed it shorts corresponding row and column. If the output line of this column is low, it makes corresponding row line low; otherwise the status of row line is high. The key is identified by data sent on the output port and input code received from the input port. The following section explains the steps required to identify pressed key.

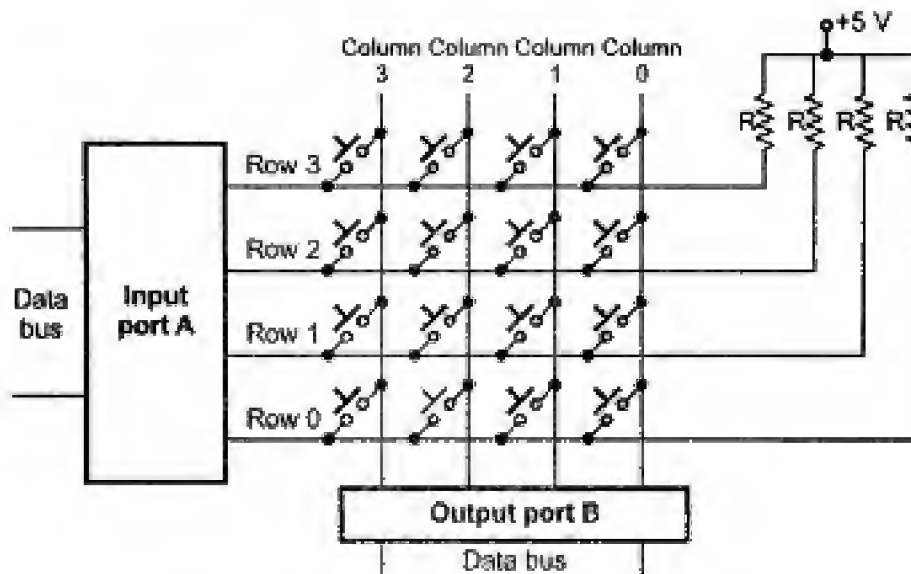


Fig. 13.18 Matrix keyboard connections

Check 1 : Whether any key is pressed or not

1. Make all column lines zero by sending low on all output lines. This activates all keys in the keyboard matrix. (Note : When scan lines are logic high, the status on the return lines do not change, it will remain logic high.)
2. Read the status of return lines. If the status of all lines is logic high, key is not pressed; otherwise key is pressed.

Check 2 :

1. Activate keys from any one column by making any one column line zero.
2. Read the status of return lines. The zero on any return line indicates key is pressed from the corresponding row and selected column. If the status of all lines is logic high, key is not pressed from that column.
3. Activate the keys from the next column and repeat 2 and 3 for all columns.

We will see how matrix keyboard can be connected to the 8051, a single chip microprocessor/microcontroller. Fig. 13.19 shows the 4 × 4 matrix keyboard connected to the port 1 of 8051. 4 lines of port 1 (P14-P17) are used as scan lines and remaining 4 lines (P10-P13) are used as return lines. The algorithm and program for keyboard interface is explained in the next section.

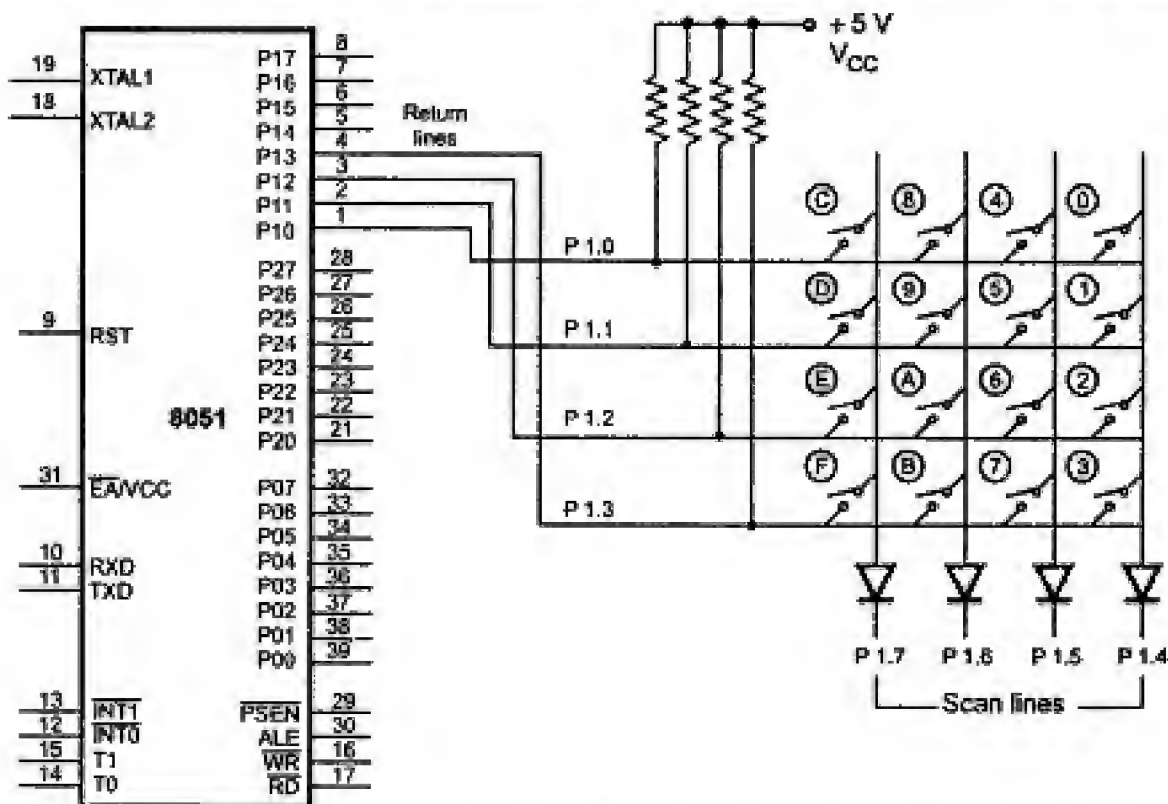


Fig. 13.19 4 × 4 matrix keyboard connected to port 1 of 8051

The steps in algorithm are as follows :

1. Initialise P1.0, P1.1, P1.2, P1.3 as inputs i.e. write '1' to these pins.
2. Check if all the keys are released by writing '0' to P1.4-P1.7 and check if all return lines are in state '1'. If No then wait.
If Yes then go to step 3.
3. Call debounce.
4. Wait for key closure. Ground all scan lines by writting '0' and then check if atleast one of return lines shows '0' level.
Key pressed ? No step 4
Yes step 5
5. Call debounce. (allow sufficient time for debounce)
6. Is key really pressed ? (Ground all scan lines by writing '0' and then check if atleast one of the return lines shows '0' level.)
No step 4
Yes step 7
7. Find key code and display the key pressed on 7-segment display.
(By grounding one scan line at a time and checking return lines for any one line to go to '0' level.)
8. Go to step 1.

Program :

```

    org lookup_table_address
    db 30h, 31h, 32h, 33h, 34h, 35h, 36h, 37h,
    38h, 39h, 41h, 42h, 43h, 44h, 45h, 46h
    org program_start_address
beg:    mov P1, #0fh      ; configure lower 4 lines of port 1
        ; as i/p
        mov dptr, #lookup_table_address ; initialise dptr with
        ; lookup_table_addr.

aga:    mov a, P1        ;
        anl a, #0fh      ;
        cjne a, #0fh, aga ; check for key released
        lcall delay      ; call delay routine for key debounce
agal:   mov a, P1        ;
        anl a, #0fh      ;
        cjne a, #0fh, go  ; check for key pressed
        ljmp agal
go:     lcall delay      ; call delay routine for key debounce
        mov a, P1
        anl a, #0fh
        cjne a, #0fh, gol ; is key really pressed ?
        ljmp agal
gol:    mov r1, #01h     ; initialise counter 1
        mov r0, #0efh   ; store word for column selection
        mov r3, #04h    ; initialise column counter
aga3:   mov P1, r0       ; select only 1 column
        mov a, P1       ; get the status of return lines
        jnb acc.0, display ; check bit 0 and if it is 0 jump
        ; to display
        inc dptr        ; increment lookup_table pointer
        jnb acc.1, display ; check bit 1 and if it is 1 jump
        ; to display
        inc dptr        ; increment lookup_table pointer
        jnb acc.2, display ; check bit 2 and if it is 2 jump
        ; to display
        inc dptr        ; increment lookup_table pointer
        jnb acc.3, display ; check bit 3 and if it is 3 jump
        ; to display
        inc dptr        ; increment lookup_table pointer
        mov a, r0       ; get the word for column selection
        rl a            ; select next column
        mov r0, a       ; store word for column selection
        djnz r3, aga3    ; check for last column
        ljmp beg        ; if key any is not pressed scan
        ; again
end

```

13.5 Interfacing Display

Most of the microprocessor/microcontroller based instruments and machines need to display letters of the alphabet and numbers to give directions or data values to users. This information can be displayed using CRT, LED or LCD displays. CRT displays are used when a large amount of data is to be displayed. In systems where only a small amount of data is to be displayed, simple LED and LCD displays are used. The following sections explain you how to interface these two types of displays to microprocessors/microcontroller.

13.5.1 LED Displays

LED displays are available in two very common formats. 7-segment displays and 5 by 7 dot-matrix displays.

Seven-Segment display

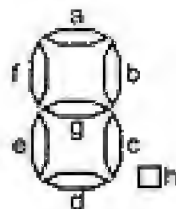


Fig. 13.20 Seven-segment display

Seven-segment displays are generally used as numerical indicators and consist of a number of LEDs arranged in seven-segments as shown in the Fig. 13.20.

Any number between 0 and 9 can be indicated by lighting the appropriate segments.

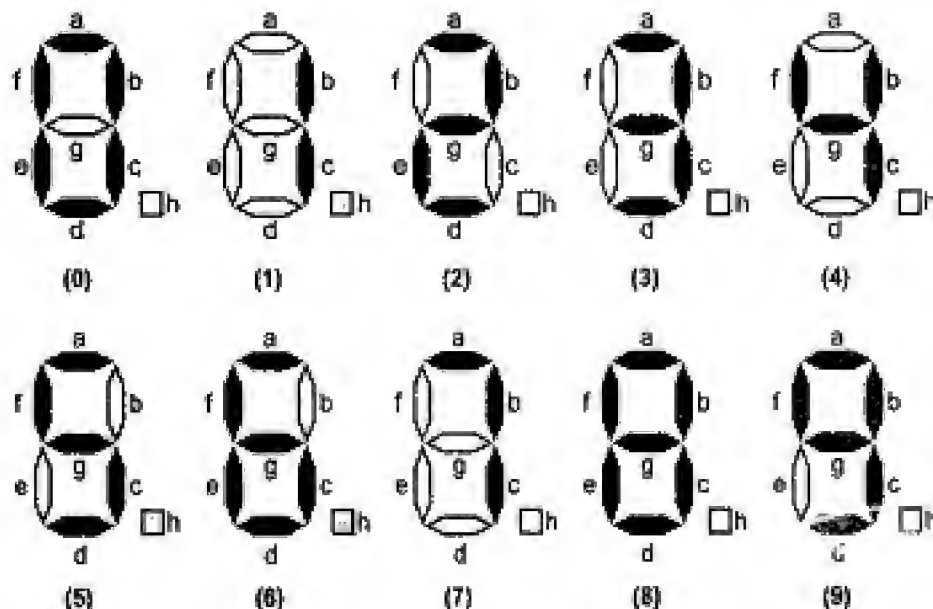


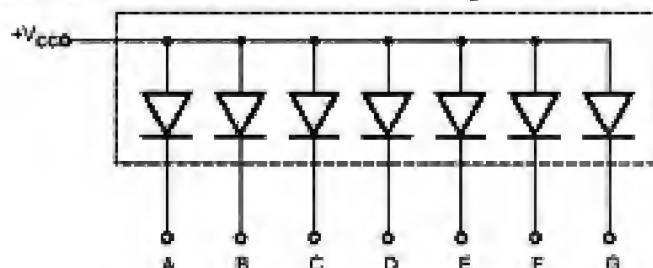
Fig. 13.21

The seven-segments are labeled a to g and dot is labelled as h. By forward biasing different LED segments, we can display the digits 0 through 9. For instance, to display 0, we need to light up a, b, c, d, e, and f. To light up 5, we need segments a, f, g, c and d.

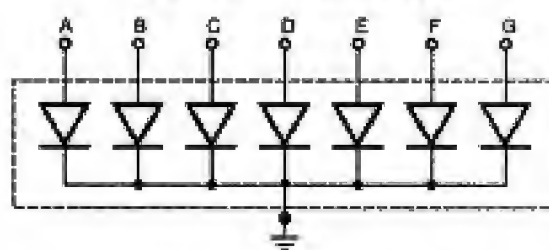
These 7-segment displays are of two types :

- Common anode type
- Common cathode type

In common anode, all anodes of LEDs are connected together as shown in Fig. 13.22 (a) and in common cathode, all cathodes are connected together, as shown in Fig. 13.22 (b).



(a) Common anode type



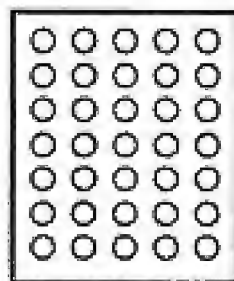
(b) Common cathode type

Fig. 13.22 Internal diagram of 7-segment LED

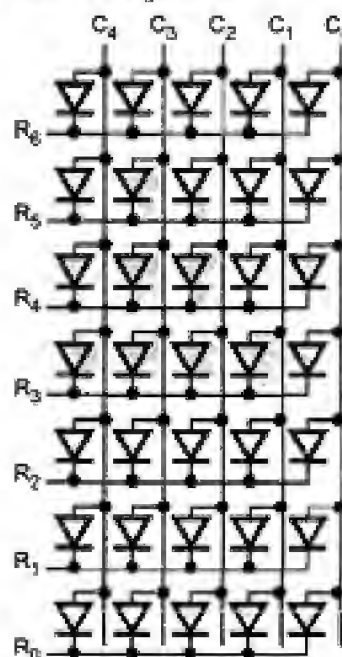
5 by 7 DOT matrix LED

Fig. 13.23 (a) and (b) show the 5 by 7 dot matrix LED display and its circuit connections.

This display can be used to display number as well as alphabets.



(a) 5 × 7 dot matrix LED display



(b) 5 × 7 dot matrix circuit connections

Fig. 13.23

13.5.2 Interfacing LED Displays

Static Display

Fig. 13.24 shows a circuit to drive a single, seven-segment, common anode LED display. For common anode, when anode is connected to positive supply, a low voltage is applied to a cathode to turn it on. Here, BCD to seven-segment decoder, IC 7447 is used to apply low voltages at cathodes according to BCD input applied to 7447. To limit the current through LED segments resistors are connected in series with the segments. This circuit connection is referred to as a static display because current is being passed through the display at all times.

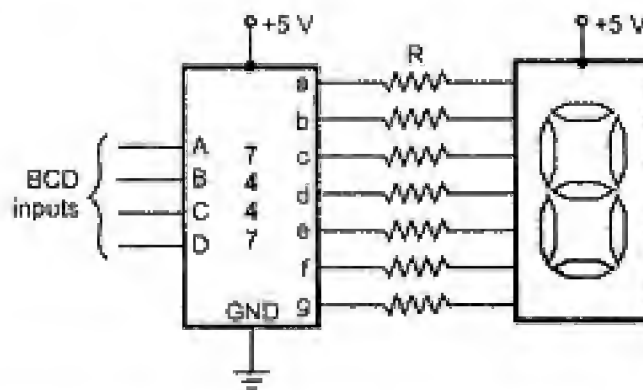


Fig. 13.24 Circuit for driving single seven-segment LED display

The value of the resistor in series with the segment can be calculated as follows :

We know, $V_{CC} - \text{drop across LED segment} - IR = 0$

Drop across LED segment is nearly 1.5 V.

$$\begin{aligned}\therefore IR &= V_{CC} - 1.5 \text{ V} \\ &= 5 - 1.5 \text{ V} \\ &= 3.5 \text{ V}\end{aligned}$$

Each LED segment requires a current of between 5 and 30 mA to light. Let's assume that current through LED segment is 15 mA.

$$\begin{aligned}\therefore R &= \frac{3.5 \text{ V}}{15 \text{ mA}} \\ &= 233 \Omega\end{aligned}$$

In practice, the voltage drop across the LED and the output of 7447 are not exactly predictable and the exact current through the LED is not critical as long as we don't exceed its maximum current rating. Therefore, a standard value 220 Ω can be used.

The static display circuits work well for driving just one or two LED digits. However, these circuits are not suitable for driving more LED digits, say 8 digits. When there are more number of digits, the first problem is power consumption. For worst case

calculations, assume that all eight digits with all segments are lit. Therefore, worst case current required is

$$\begin{aligned} I &= 8 \text{ (digits)} \times 7 \text{ (segment)} \\ &\quad \times 15 \text{ mA (current per segment)} \\ &= 840 \text{ mA} \end{aligned}$$

A second problem of the static approach is that each display digit requires a separate BCD to 7-segment decoder.

Multiplexed Display

To solve the problems of the static display approach, multiplexed display method is used. Fig. 13.25 shows the 4 seven-segment displays connected using multiplexed method. Here, common anode seven-segment LEDs are used.

Anodes are connected to +5 V through transistors. Cathodes of all seven-segments are connected in parallel and then to the output of 7447 IC through resistors. Looking at the Fig. 13.25, the question may occur in our mind that, "Aren't all of the digits going to

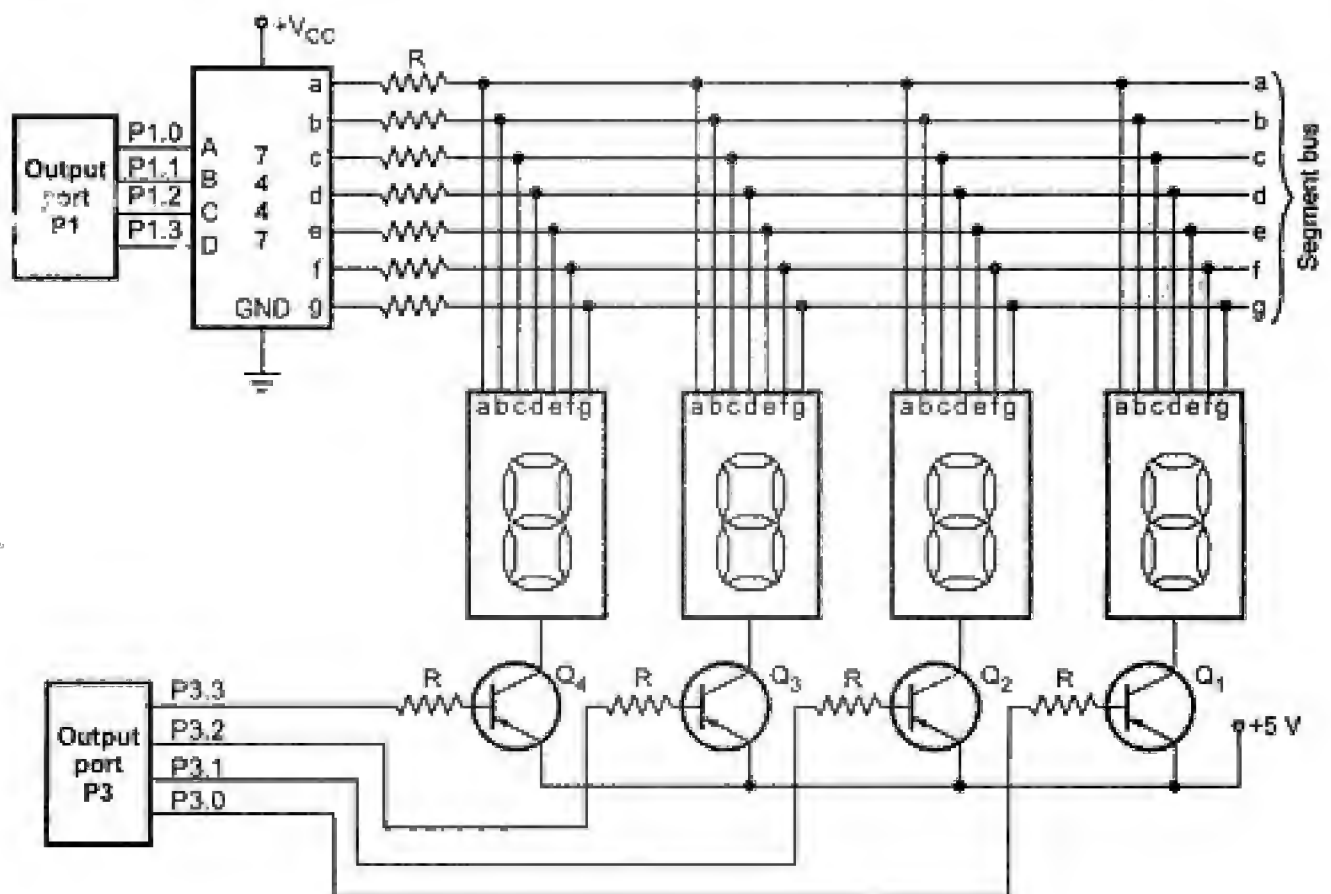


Fig. 13.25 Seven-segment display in multiplexed connection

display the same number?" The answer is that they would show the same number only if all the digits are turned on at the same time. However, in multiplexed display the segment information is sent for all digits on the common lines (output lines of 7447), but only one display digit is turned on at a time. The PNP transistors connected in series with the common anode of each digit act as an ON and OFF switch for that digit. Here's how the multiplexing process works.

The BCD code for digit 1 is first output from port 1, to the 7447. The 7447, BCD to seven-segment decoder outputs the corresponding seven-segment code on the segment bus lines. The transistor Q_1 connected to digit 1 is then turned on by outputting a low to that bit of port 3. All of the rest of the bits of port 3 are made high to ensure no other digits are turned on. After 2 ms, digit 1 is turned OFF outputting all highs to port 3. The BCD code for digit 2 is then output to the port 1, and bit pattern to turn on digit 2 is output on port 3. After 2 ms, digit 2 is turned off and the process is repeated for digit 3 and digit 4. After completion of turn for each digit, all the digits are lit again in turn.

With 4 digits and 2 ms per digit we get back to digit 1 every 8 ms or about 125 times a second. This refresh rate is fast enough that, to our eye and due to persistence of vision all digits will appear to be lit all the time.

In multiplexed display, the segment current is kept in between 40 mA to 60 mA so that they will appear as bright as they would if not multiplexed. Even with this increased segment current, multiplexing gives a large saving in power and hardware components.

➡ **Example 13.2 :** *Interface an 8-bit 7-segment LED display to 8051 through port 1 and port 3 and write an 8051 assembly language program to display message on the display.*

Solution :

Hardware

The Fig. 13.26 shows the multiplexed 8-digit 7-segment LED display connected in 8051 system using port 1 and port 3. In this circuit port 1 and port 3 are used as a latch, i.e. output port. Port 1 provides the segment data inputs to the display and port 3 provides a means of selecting a display position at a time for multiplexing the displays. Here, instead of BCD to seven-segment decoder (IC 7447) transistors are used to drive the LED segments. Due to this we can also display HEX characters on the display; however in this case we have to send the proper 7-segment code of a particular digit that is to be displayed on the port 1.

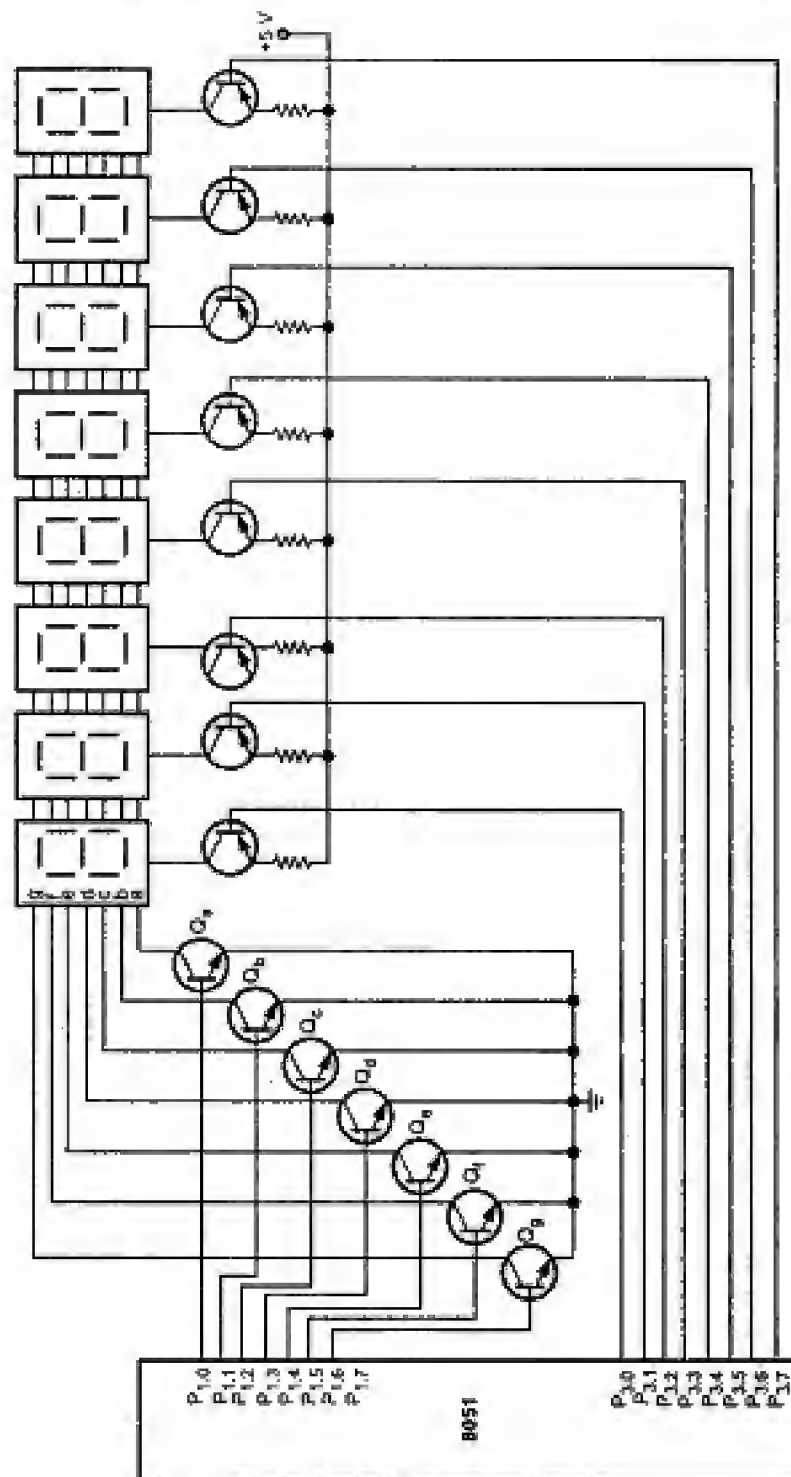


Fig. 13.26

Subroutine to Display Message

```

MOV R0, #8H      ; Initialize counter
MOV R1, #7FH     ; load select pattern
MOV DPTR, #6000H ; Starting address of message to
                  ; be displayed

```

```
AGAIN:  MOV P3, R1      ; select digit
        MOVX A, @DPTR   ; Get data
        MOV P1, A       ; Send data
        LCALL DELAY     ; Wait for some time
        MOV A, R1       ; [ Adjust
        RR A            ; selection
        MOV R1, A       ; pattern]
        INC DPTR        ; Increment message pointer
        DJNZ R0, AGAIN  ; Decrement R0 and check
        RET             ; if for zero; if not, goto AGAIN
```

Note :

This subroutine must be called continuously to display the 7-segment coded message stored in the memory from address 6000H.

13.6 Interfacing LCD Display

The liquid crystals are one of the most fascinating material systems in nature, having properties of liquids as well as of a solid crystal. The terms liquid crystal refers to the fact that these compounds have a crystalline arrangement of molecules, yet they flow like a liquid. Liquid crystal displays do not emit or generate light, but rather alter externally generated illumination. Their ability to modulate light when electrical signal is applied has made them very useful in flat panel display technology.

There are two types of liquid crystal displays (LCDs) according to the theory of operation : 1. Dynamic scattering 2. Field effect.

Fig. 13.27 shows the construction of a typical liquid crystal display. It consists of two glass plates with a liquid crystal fluid in between. The back plate is coated with thin transparent layer of conductive material, whereas front plate has a photoetched conductive coating with seven-segment pattern as shown in Fig. 13.27.

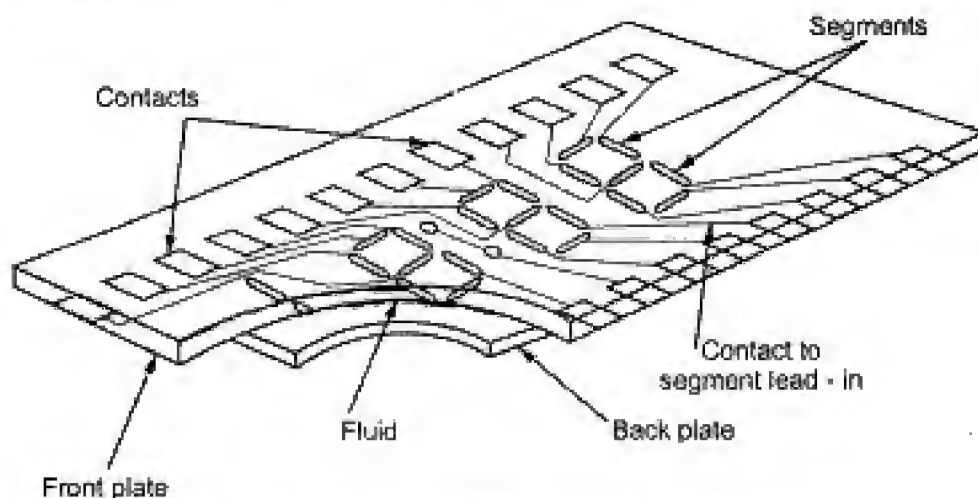


Fig. 13.27 Liquid crystal display construction

In the absence of the electrical signal, orientation order is maintained in the crystal allowing light to transmit. This makes LCD display clear. The current through the liquid crystal causes orientation order to collapse. The random orientation results scattering of light which lights display segment on a dark background.

Fig. 13.28 shows the circuit for driving LCD seven-segment display using IC 4543B.

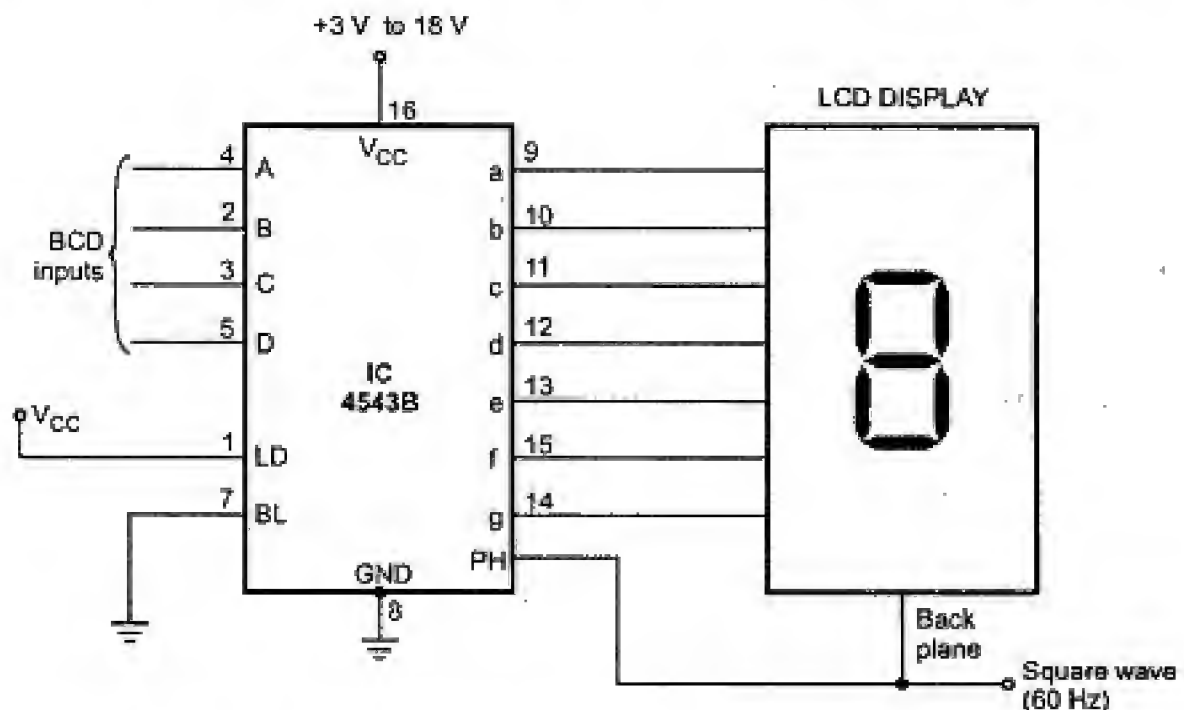


Fig. 13.28 Circuit for driving LCD seven segment display using 4543B

The 4543B BCD to 7-segment latch/decoder/driver is designed for liquid crystal displays. Pins A, B, C and D represent BCD inputs with A as a least significant bit (LSB) and D as a most significant bit (MSB). Pins a through g are the seven-segment outputs. The 4543B has three control terminals : LD (Latch Disable), PH (Phase), and BL (Blank). In normal use the LD terminal is held high and BL terminal is tied low. The state of the PH terminal depends on the type of display that is being driven. For driving LCD displays, a square wave (about 60 Hz swinging fully between the GND and V_{CC} values) must be applied to the phase terminal.

The display can be blanked by simply driving the BL terminal to the logic high state. When the LD terminal is in its normal high state, BCD inputs are decoded and fed directly to the seven-segment output terminals of the IC. When the LD terminal is pulled low, the BCD input signals that are present at the moment of transition are latched into memory and fed to the seven-segment outputs.

The Fig. 13.29 shows how above circuit can be used to drive a 4-digit nonmultiplexed, 7-segment LCD display. Here, BCD input for each display is latched in the corresponding latch. The latch enable signals are activated using 2 : 4 decoder in synchronisation with the BCD inputs.

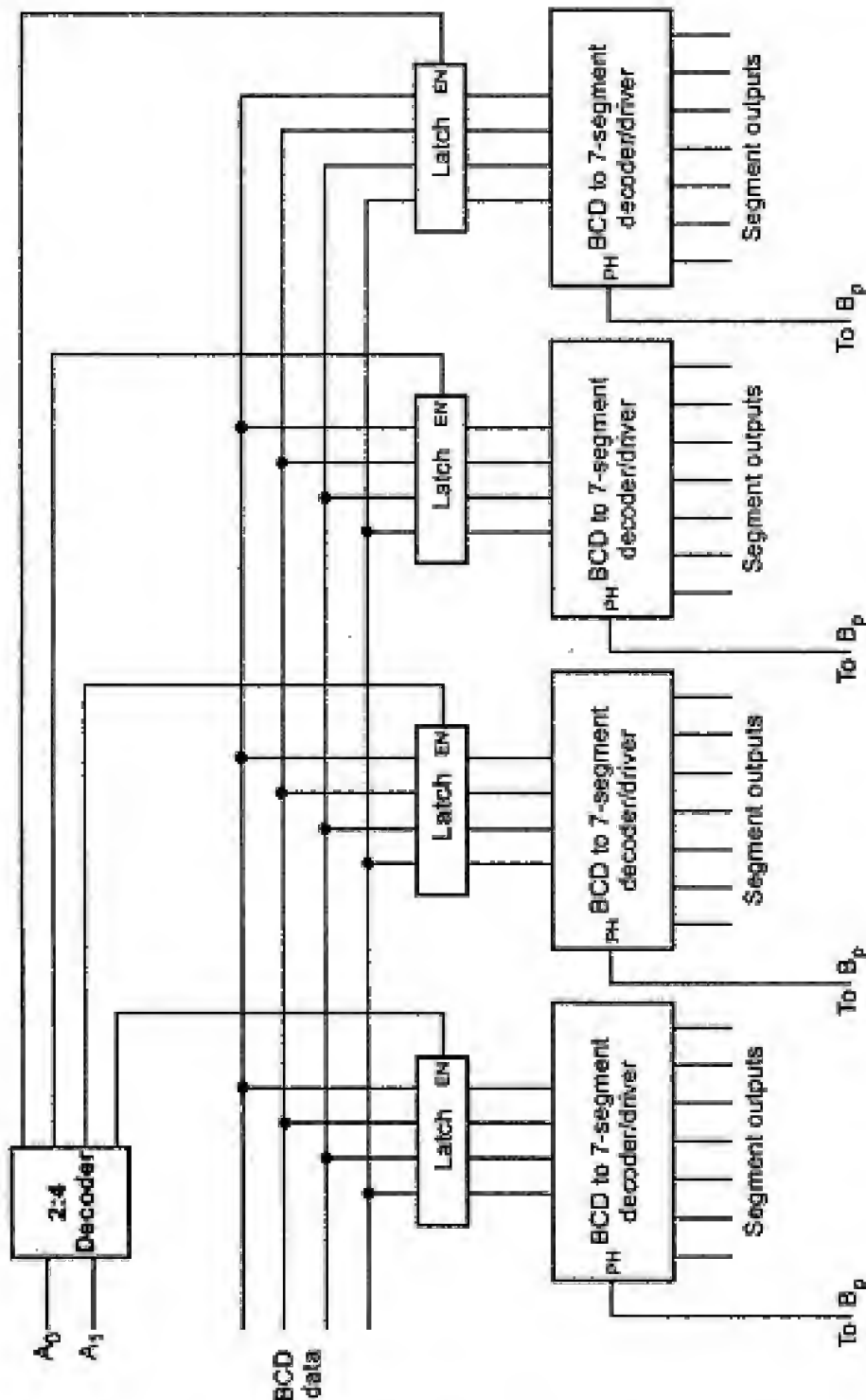


Fig. 13.29 4-7 segment LCD display driving circuit

Now-a-days, many LCD modules are available which have built-in drivers for LCD and interfacing circuitry to interface them to microprocessor/microcontroller systems. These LCD modules allow display of characters as well as numbers. They are available in 16×2 , 20×1 , 20×2 , 20×4 and 40×2 sizes. The first figure represents number of character in each line and second figure represents number of lines the display has. In this section, we see the interfacing of 20×2 LCD display module to microprocessor/microcontroller system through 8255. In this module the display is organized as two lines, each of 20 characters. The module has 14-pins. The function of each pin is given in the Table 13.9.

| Pin | Symbol | I/O | Description |
|------|----------------------------------|-----|---|
| 1 | V_{SS} | - | Ground |
| 2 | V_{CC} | - | + 5 V power supply |
| 3 | V_{EE} | - | Used for controlling LCD contrast |
| 4 | RS | I | Register select input is used to select either of the two available registers in the module : Data register or command register. When RS = 0 Data register is selected. When RS = 1 Command register is selected. |
| 5 | $\overline{R/W}$ | I | Allows the user to write information to the LCD or read information from it. For reading $\overline{R/W} = 1$ and for writing $\overline{R/W} = 0$ |
| 6 | E | I | This pin is used by LCD to latch information available at its data pins. |
| 7-14 | DB ₀ -DB ₇ | I/O | This 8-bit data bus is used to send information to the LCD or read the contents of the internal registers of the LCD. |

Table 13.9 Pin description for LCD module

The Fig. 13.30 shows the interfacing of a 20 character \times 2 line LCD module with the 8051. As shown in the Fig. 13.30, the data lines are connected to the port 1 of 8051 and control lines RS, $\overline{R/W}$ and E are driven by 3.2, 3.3 and 3.1 lines of port 3, respectively. The voltage at V_{EE} pin is adjusted by a potentiometer to adjust the contrast of the LCD.

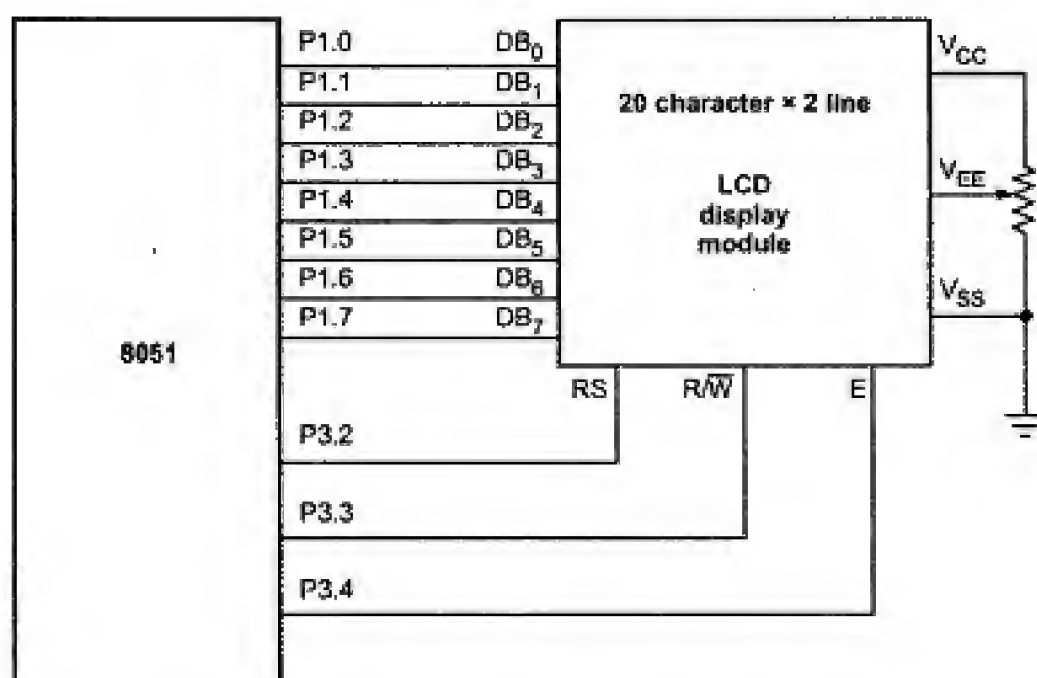


Fig. 13.30 Interfacing LCD module with 8051

The display can be controlled by issuing proper commands to the LCD module. The Table 13.10 lists the command available for LCD module.

| Command | Command Code | | | | | | | | | | Description |
|--------------------------|--------------|-----|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|--|
| | RS | R/W | DB ₇ | DB ₆ | DB ₅ | DB ₄ | DB ₃ | DB ₂ | DB ₁ | DB ₀ | |
| Clear display | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | Sets DD RAM address to 0 and clears entire display. |
| Return home | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | - | Sets DD RAM address to 0 and returns the cursor to the home position. DD RAM contents remain unchanged. |
| Entry mode set | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1/D | S | Sets cursor move direction and specifies or not to shift the display. These operations are performed during data write and read. |
| Display ON/OFF control | 0 | 0 | 0 | 0 | 0 | 0 | 1 | D | C | B | Sets ON/OFF of entire display (D), cursor ON/OFF (C), and blink of cursor position character (B). |
| Cursor and display shift | 0 | 0 | 0 | 0 | 0 | 1 | S/C | R/L | - | - | Moves the cursor and shifts the display without changing DD RAM contents. |

for that RAM by issuing proper command. Then we can write data into it by activating write cycle. To activate write cycle we have to make R/W signal low, RS signal high, send data on port A and apply high to low pulse of atleast 450 ns of duration on E pin of the module. The DD RAM stores the characters in their ASCII code whereas CG RAM stores the character in its internally generated character code. Let us see the 8051 assembly language program to display 'WELCOME' message on the LCD module. Before sending command or data it is necessary to check busy flag, i.e. whether LCD is reading or not.

Main Routine :

```

MOV 81H,#30H      ; Initialise stack pointer
MOV A,#3CH         ; [Send command code to set font
                    ; = 5 × 10 dots,
                    ; DL = 8-bits and N = 2 lines].
LCALL COMMAND
MOV A,#0EH         ; [Send command code to set display
                    ; and cursor ON]
LCALL COMMAND
MOV A,#01H         ; [Send command code to
                    ; clear LCD]
LCALL COMMAND
MOV A,#86H         ; [Send command to set DD RAM
                    ; address to the seventh location]
LCALL COMMAND
MOV A,#'W'
LCALL DISPLAY      ; Display letter W
MOV A,#'E'
LCALL DISPLAY      ; Display letter E
MOV A,#'L'
LCALL DISPLAY      ; Display letter L
MOV A,#'C'
LCALL DISPLAY      ; Display letter C
MOV A,#'O'
LCALL DISPLAY      ; Display letter O
MOV A,#'M'
LCALL DISPLAY      ; Display letter M
MOV A,#'E'
LCALL DISPLAY      ; Display letter E
SJMP HERE:HERE     ; Loop here after displaying
                    ; message

```

COMMAND Routine :

```

LCALL READY        ; Check whether LCD is ready ?
MOV P1, A          ; Issue command code
CLR P3.2           ; Make RS = 0 to issue command
CLR P3.3           ; Make R/W = 0 to enable writing
SETB P3.4          ; Make E = 1
CLR P3.4           ; Make E = 0
RET               ; Return

```


READY Routine :

```

CLR P3.4      ; Disable display
CLR P3.2      ; Make RS = 0 to access command register
MOV P1,#0FFH ; Configure P1 as an input port
SETB P3.3     ; Make R/W = 1 to enable reading
READ: SETB P3.4 ; Make E = 1
JB P1.7,READ  ; Check D87 bit. If it is 1, LCD is busy
              ; hence check if until it is 0
CLR P3.4      ; Make E = 0 to disable display
RET           ; Return

```

DISPLAY Routine :

```

LCALL READY ; Check whether LCD is ready ?
MOV P1, A   ; Issue data
SETB P3.2   ; Make RS = 1 to issue data
CLR P3.3    ; Make R/W = 0 to enable writing
SETB P3.4   ; Make E = 1
CLR P3.4    ; Make E = 0
RET         ; Return

```

13.7 Interfacing DAC to 8051

In this section, we are going to see the interfacing of DAC 1408, 8-bit R/2R ladder type DAC with 8051. We begin with the details of IC DAC 1408.

13.7.1 IC DAC 1408

The 1408 is an 8-bit R/2R ladder type D/A converter compatible with TTL and CMOS logic. It is designed to use where the output current is linear product of an eight-bit digital word.

Fig. 13.31 shows the pin diagram and block diagram for IC 1408 DAC.

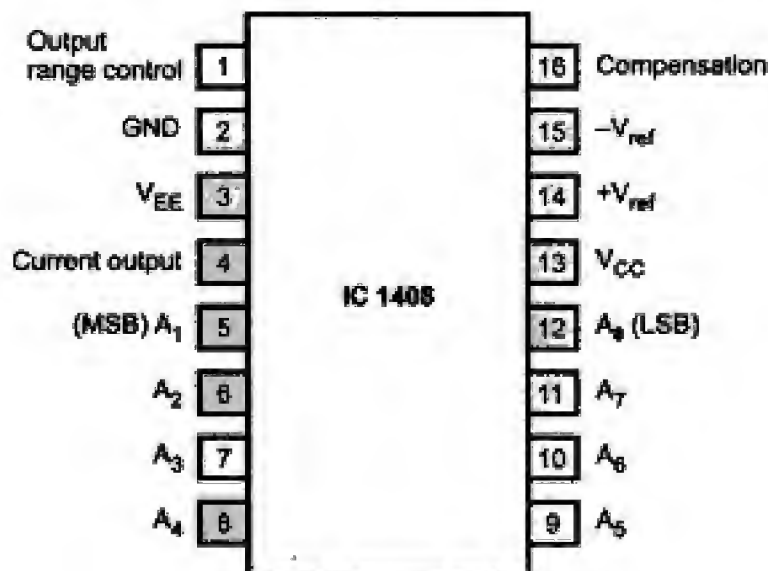


Fig. 13.31 (a) Pin diagram

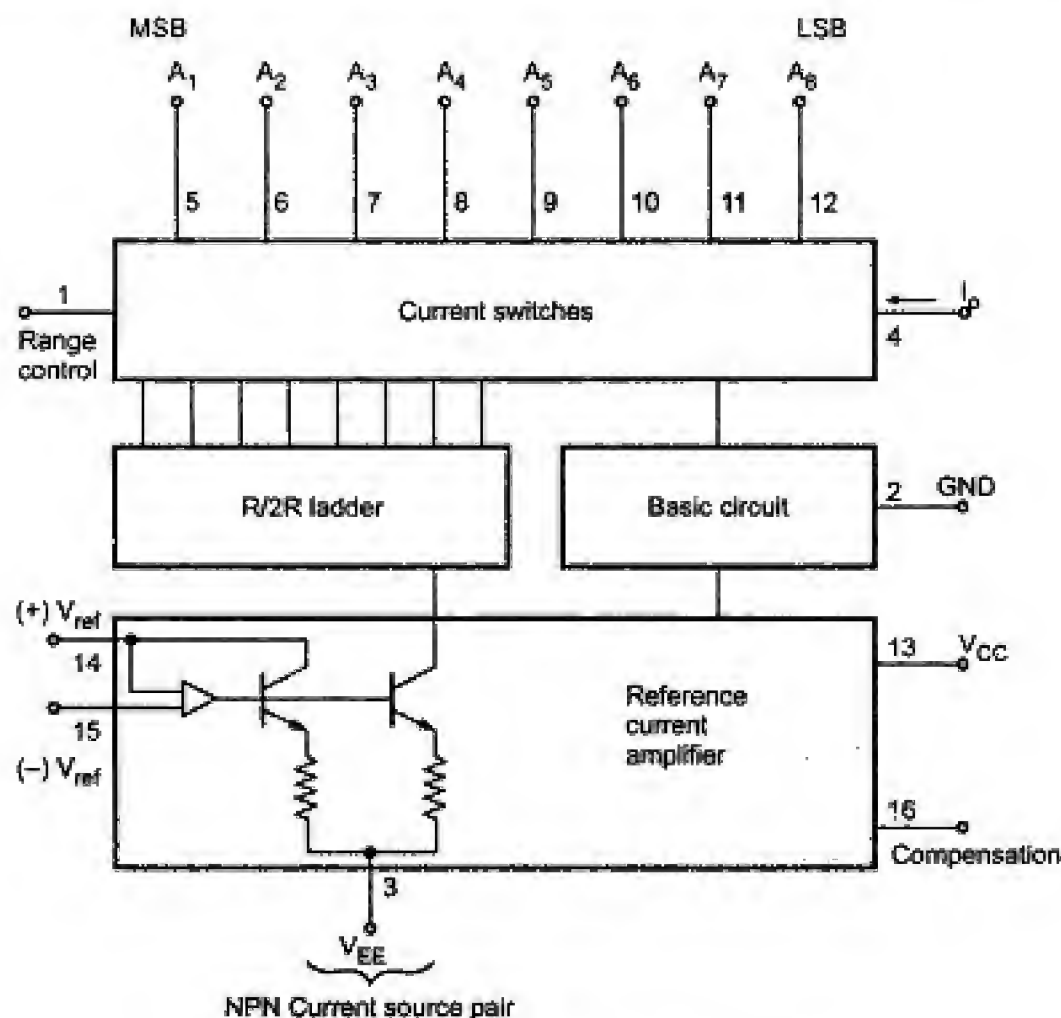


Fig. 13.31 (b) Block diagram

The IC 1408 consists of a reference current amplifier, an R/2R ladder and eight high speed current switches. It has eight input data lines A_1 (MSB) through A_8 (LSB) which control the positions of current switches.

It requires 2 mA reference current for full scale input and two power supplies $V_{CC} = +5\text{ V}$ and $V_{EE} = -15\text{ V}$ (V_{EE} can range from -5 V to -15 V).

The voltage V_{ref} and resistor R_{14} determines the total reference current source and R_{15} is generally equal to R_{14} to match the input impedance of the reference current amplifier.

Fig. 13.32 shows a typical circuit for IC 1408.

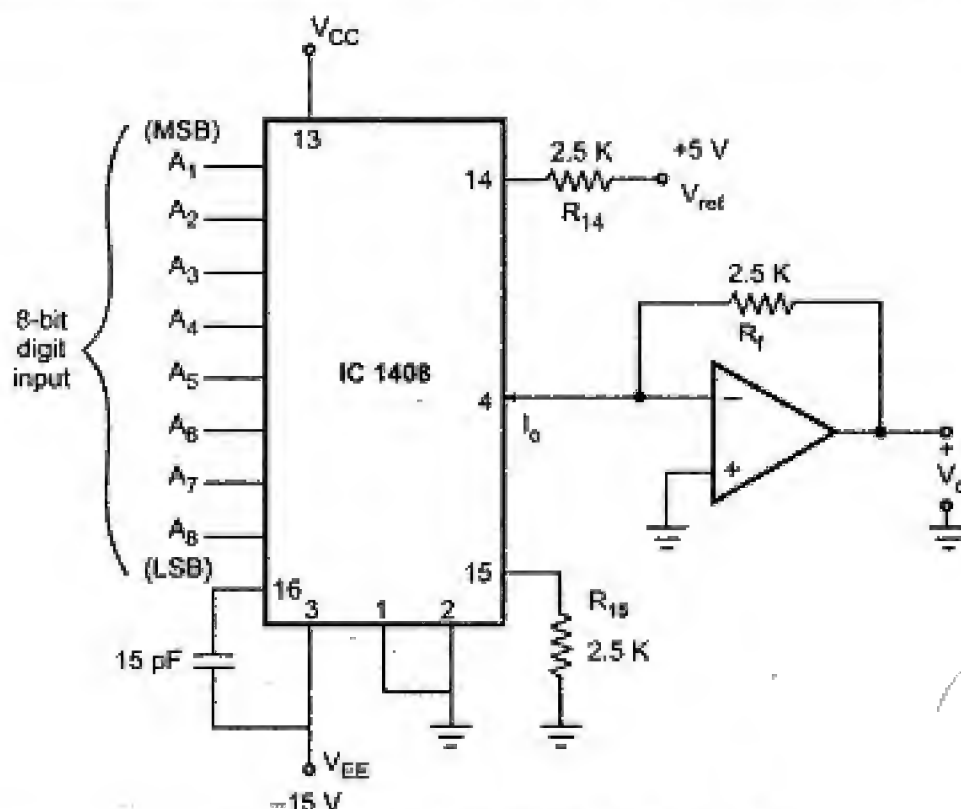


Fig. 13.32 Typical circuit for IC 1408

The output current I_o can be given as

$$I_o = \frac{V_{ref}}{R_{14}} \left(\frac{A_1}{2} + \frac{A_2}{4} + \frac{A_3}{8} + \frac{A_4}{16} + \frac{A_5}{32} + \frac{A_6}{64} + \frac{A_7}{128} + \frac{A_8}{256} \right)$$

Note : Input A_1 through A_8 can be either 0 or 1. Therefore, for typical circuit full scale current can be given as,

$$\begin{aligned} I_o &= \frac{5}{2.5 \text{ K}} \left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{32} + \frac{1}{64} + \frac{1}{128} + \frac{1}{256} \right) \\ &= \frac{2 \text{ mA} \times 255}{256} = 1.992 \text{ mA} \end{aligned}$$

It shows that the full scale output current is always 1 LSB less than the reference current source of 2 mA. This output current is converted into voltage by I to V converter. The output voltage for full scale input can be given as,

$$V_o = 1.992 \times 2.5 \text{ K} = 4.98 \text{ V}$$

Note : The arrow on the pin 4 shows the output current direction. It is inward. This means that IC 1408 sinks current. At $(0000\ 0000)_2$ binary input it sinks zero current and at $(1111\ 1111)_2$ binary input it sinks 1.992 mA.

The circuit shown in the Fig. 13.32 gives output in the unipolar range. When digital input is 00H, the output voltage is 0 V and when digital input is FFH $(1111\ 1111)_2$, the output voltage is + 5 V. This circuit can be modified to give bipolar output.

Fig. 13.33 shows the circuit for giving output in the bipolar range. Here, resistor R_B (5 K) is connected between V_{ref} and the output terminal of IC 1408. This gives a constant current source of 1 mA.

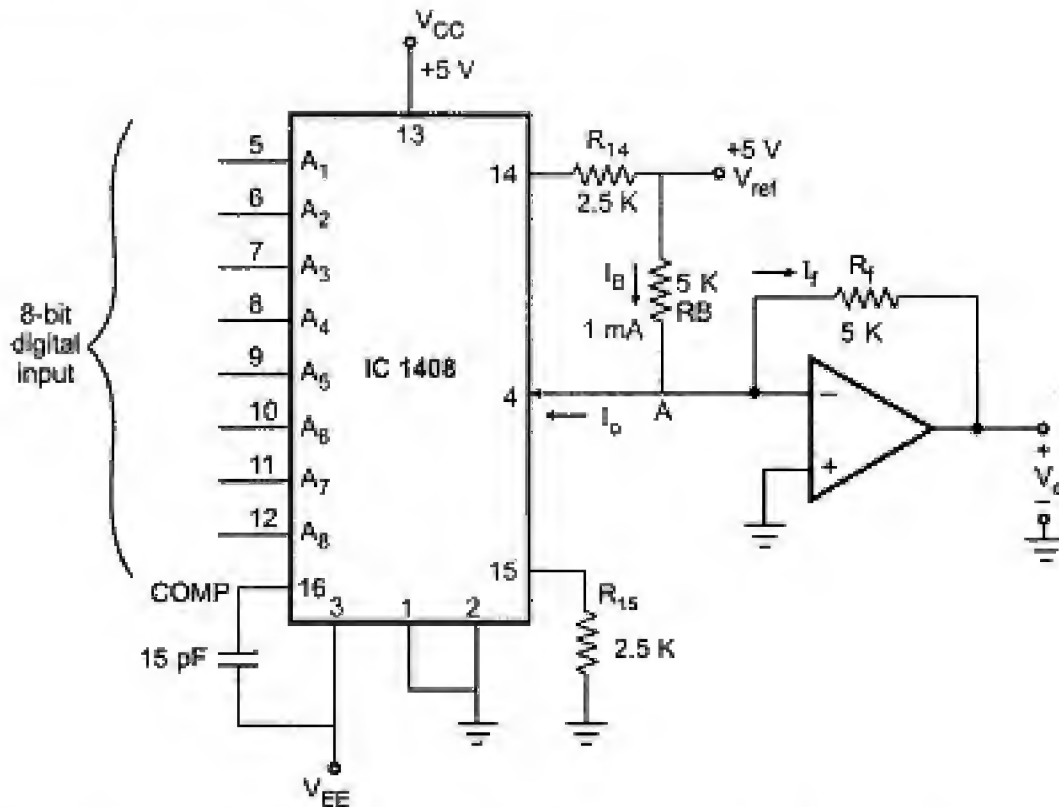


Fig. 13.33 Interfacing DAC in the bipolar range

The circuit operation can be observed for three conditions :

Condition 1 : For binary input 00H)

When binary input is 00H, the output current I_o at pin 4 is zero. Due to this current flowing through R_B (1 mA) flows through R_f giving $V_o = -5$ V.

Condition 2 : For binary input 80H

When binary input is 80H, the output current I_o at pin 4 is 1 mA. By applying KCL at node A we get,

$$-I_B + I_o + I_f = 0$$

Substituting values of I_B and I_o we get,

$$-(1 \text{ mA}) + (1 \text{ mA}) + I_f = 0$$

$$\therefore I_f = 0$$

and therefore the output voltage is zero.

Condition 3 : For binary input FFH

When binary input is FFH, the output current I_o at pin 4 is 2 mA. By applying KCL at node A we get,

$$-I_B + I_o + I_f = 0$$

substituting values of I_B and I_o we get,

$$-(1 \text{ mA}) + (2 \text{ mA}) + I_f = 0$$

$$\therefore I_f = -1 \text{ mA}$$

Therefore, the output voltage is + 5 V. In this way, circuit shown in the Fig. 13.33 gives output in the bipolar range.

13.7.1.1 Important Electrical Characteristics for IC 1408

- Reference current : 2 mA
- Supply voltage : + 5 V_{CC} and - 15 V V_{EE}
- Setting time : 300 ns
- Full scale output current : 1.992 mA
- Accuracy : 0.19 %

13.7.2 Interfacing DAC 1408 / DAC 0808 with 8051

Fig. 13.34 shows the interfacing of DAC 0808 with 8051 microcontroller based system.

We now see how different waveforms can be generated using this circuit.

Square Wave

To generate square wave first we have to output FF and then 00 on port 1 of 8051. The port 1 is connected as an input to the DAC 0808. According to frequency requirement delay is provided between the two outputs.

Program :

```

                                MOV SP, #08H           ; Initialize Stack Pointer
REPEAT:                        MOV PI, #0FFH           ; Load all 1's in Port 1
                                LCALL DELAY             ; Call delay routine
                                MOV P, #00H            ; Load all 0's in Port 1
                                LCALL DELAY             ; Call delay routine
                                LJMP REPEAT             ; Repeat
DELAY:                          MOV R0, #0FFH           ; Load delay count
BACK:                          DJNZ R0, BACK           ; Decrement and check whether delay
                                ; count is zero if not repeat the
                                ; operation
                                RET                     ; Return to main program

```

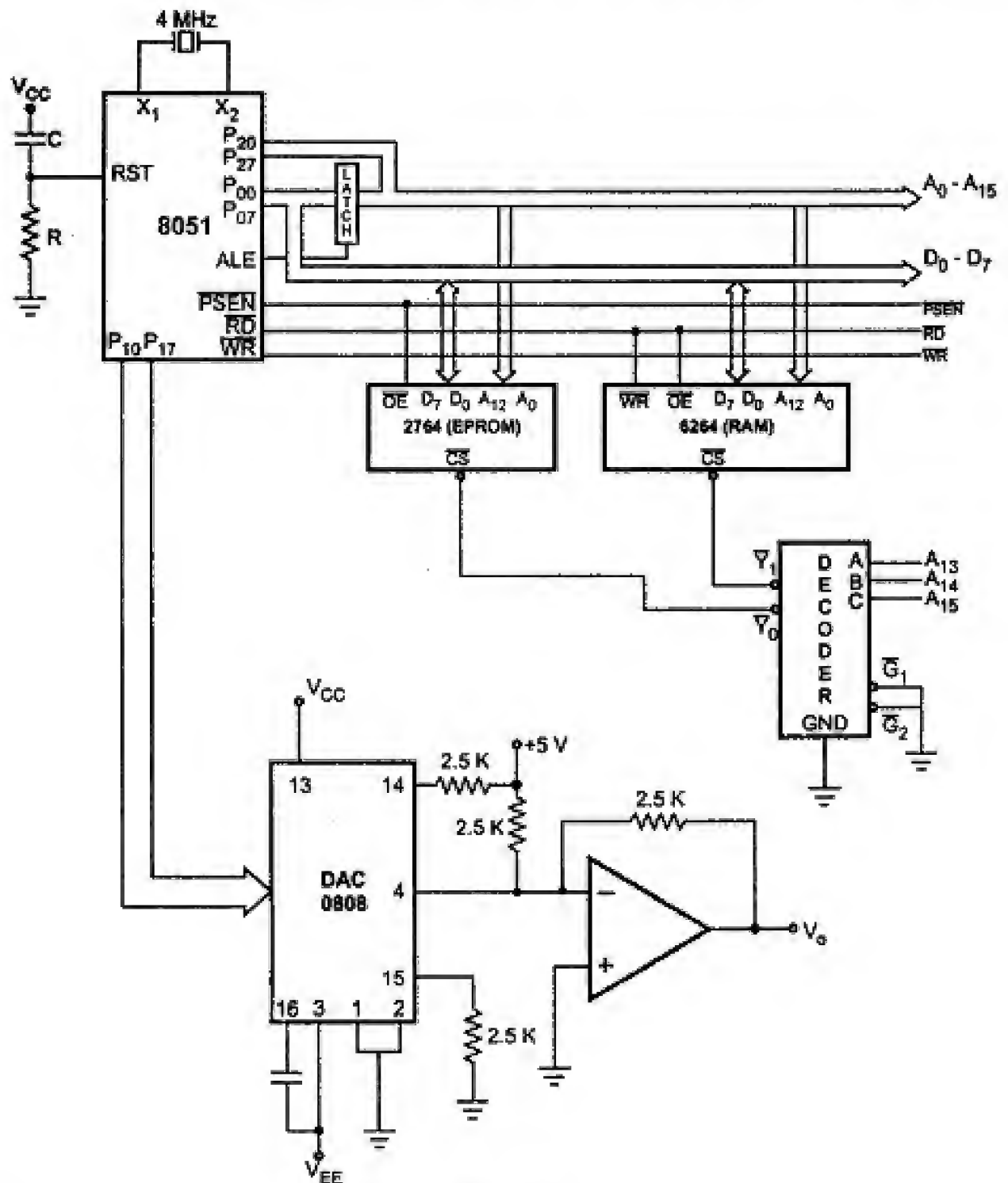


Fig. 13.34

Triangular Wave

To generate triangular wave we have to output data from 00 initially, and it should be incremented upto FF. When it reaches FF it should be decremented upto 00.


```

Program :      M
                MOV SP, #08H           ; Initialize Stack Pointer
                MOV R0, #00H
REPEAT:        MOV P1, R0              ; Send digital data to the input
                ; of DAC 0808
                INC R0                  ; Increment digital data
                CJNE R0, #0FFH, REPEAT  ; Check digital data for peak
                ; output if not repeat
REPEAT1:       MOV P1, R0              ; Send digital data to the input
                ; of DAC 0808
                DJNZ R0, REPEAT1        ; Decrement digital data and
                ; check digital data for least
                ; output if not repeat.
                LJMPL REPEAT

```

Sine Wave Generate :

To generate sine wave we have to output digital equivalent values which will represent sine wave as shown in the Fig. 13.35. Digital data 00H represents -2.5 V, 7FH represents 0 V and FFH represents $+2.5$ V. The digital equivalent for sine wave can be calculated as follows.

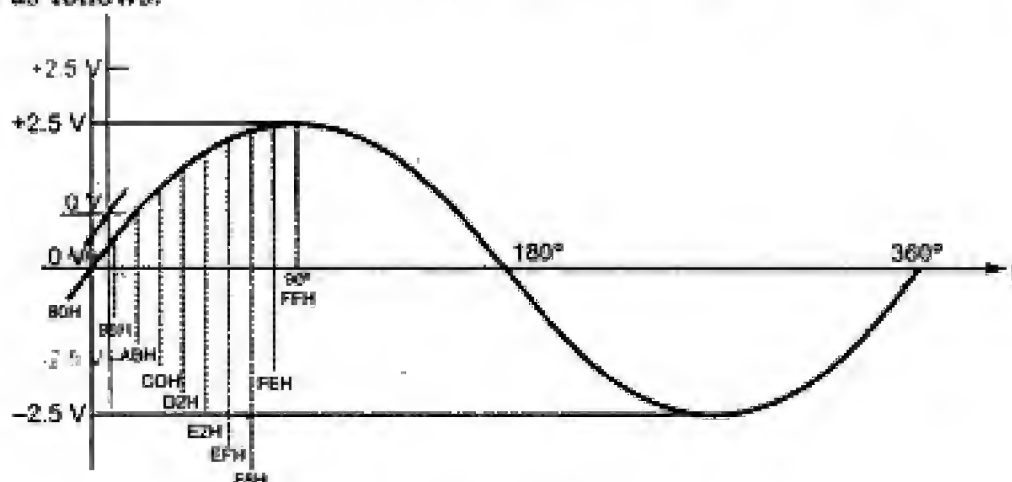


Fig. 13.35

We know that $\sin 0^\circ = 0$ and $\sin 90^\circ = 1$. The range $\sin 0^\circ$ to $\sin 90^\circ$ is distributed over digital range of 7FH to FFH i.e. $(FFH - 7FH)$ 128 decimal steps. Therefore, taking 128 as a offset we can write,

Digital equivalent value (DEV) for $\sin x = (128 + 128 \times \sin x)$

where x is angle in degrees and digital value is in decimal.

Lookup table shows the digital equivalent values for sine wave.

| Degrees | Equation | Digital Equivalent in Decimal | Digital Equivalent in Hex |
|---------|-------------------------------|-------------------------------|---------------------------|
| 0° | $(128 + 128 \times \sin 0)$ | 128 | 80H |
| 10° | $(128 + 128 \times \sin 10)$ | 150 | 96H |
| 20° | $(128 + 128 \times \sin 20)$ | 171 | ABH |
| 30° | $(128 + 128 \times \sin 30)$ | 192 | C0H |
| 40° | $(128 + 128 \times \sin 40)$ | 156 | D2H |
| 50° | $(128 + 128 \times \sin 50)$ | 226 | E2H |
| 60° | $(128 + 128 \times \sin 60)$ | 239 | EFH |
| 70° | $(128 + 128 \times \sin 70)$ | 248 | F8H |
| 80° | $(128 + 128 \times \sin 80)$ | 254 | FEH |
| 90° | $(128 + 128 \times \sin 90)$ | 256 → 255 | FFH |
| 100° | $(128 + 128 \times \sin 100)$ | 254 | FEH |
| 110° | $(128 + 128 \times \sin 110)$ | 248 | F8H |
| 120° | $(128 + 128 \times \sin 120)$ | 239 | EFH |
| 130° | $(128 + 128 \times \sin 130)$ | 226 | E2H |
| 140° | $(128 + 128 \times \sin 140)$ | 156 | D2H |
| 150° | $(128 + 128 \times \sin 150)$ | 192 | C0H |
| 160° | $(128 + 128 \times \sin 160)$ | 171 | ABH |
| 170° | $(128 + 128 \times \sin 170)$ | 150 | 96H |
| 180° | $(128 + 128 \times \sin 180)$ | 128 | 80H |
| 190° | $(128 + 128 \times \sin 190)$ | 106 | 6AH |
| 200° | $(128 + 128 \times \sin 200)$ | 84 | 54H |
| 210° | $(128 + 128 \times \sin 210)$ | 64 | 40H |
| 220° | $(128 + 128 \times \sin 220)$ | 46 | 2EH |
| 230° | $(128 + 128 \times \sin 230)$ | 30 | 1EH |
| 240° | $(128 + 128 \times \sin 240)$ | 17 | 11H |
| 250° | $(128 + 128 \times \sin 250)$ | 08 | 08H |
| 260° | $(128 + 128 \times \sin 260)$ | 02 | 02H |
| 270° | $(128 + 128 \times \sin 270)$ | 00 | 00H |
| 280° | $(128 + 128 \times \sin 280)$ | 02 | 02H |
| 290° | $(128 + 128 \times \sin 290)$ | 08 | 08H |
| 300° | $(128 + 128 \times \sin 300)$ | 17 | 11H |
| 310° | $(128 + 128 \times \sin 310)$ | 30 | 1EH |
| 320° | $(128 + 128 \times \sin 320)$ | 46 | 2EH |
| 330° | $(128 + 128 \times \sin 330)$ | 64 | 40H |
| 340° | $(128 + 128 \times \sin 340)$ | 84 | 54H |
| 350° | $(128 + 128 \times \sin 350)$ | 106 | 6AH |
| 360° | $(128 + 128 \times \sin 360)$ | 128 | 80H |

Table 13.11

```

AGAIN:  MOV DPTR, #LUT      ; Initialize pointer to lookup table
        MOV R0, #25H       ; Initialize counter
BEGIN:  CLR A
        MOVC A, @A+DPTR    ; Get data from lookup table
        MOV P1, A          ; Send data to port 1
        INC DPTR           ; point to next lookup table entry
        DJNZ R0, BEGIN     ; Decrement counter and go to BEGIN
                        ; if not zero
        SJMP AGAIN        ; Repeat the entire process
        ORG 300
LUT:    DB 80H, 96H, ABH, . . . . .
        DB . . . . .
        DB . . . . . 40H, 54H, 6AH, 80H

```

Note :

In above program we have to store entries from given lookup table instead of dots.

13.8 Interfacing ADC to 8051

In this section, we are going to see the interfacing of ADC 0803/0805 and 0808/0809 with 8051. We begin with the details of IC ADC 0803/0805 and ADC 0808/0809.

13.8.1 ADC 0804 Family

The ADC 0803, ADC 0804 and ADC 0805 are CMOS 8-bit successive-approximation analog to digital converters. These devices are design to operate from common microprocessor control buses, with tri-state output latches driving the data bus, and are identical except for accuracy.

Pin Diagram

Fig. 13.36 shows the pin diagram of ADC 0803, ADC 0804 and ADC 0805. IN+ and IN- inputs allow application of differential input voltage which has high common mode rejection and eliminates offset due to the zero input analog voltage value. The devices can

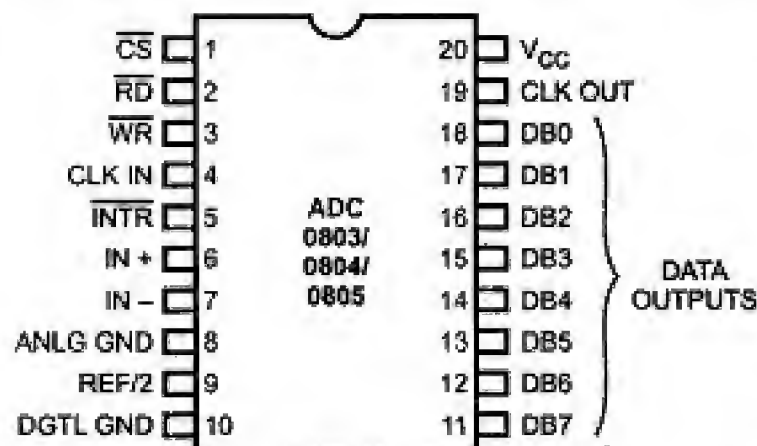


Fig. 13.36 Pin diagram of ADC 0803/0804/0805

operate with an external clock signal or, the on chip clock generator can be used independently by adding an external resistor and capacitor to set the time period.

Features

- 8-bit successive approximation ADC
- Conversion time 100 μ s
- Access time 135 ns.
- It has an on-chip clock generator.
- It does not require any zero adjustment.
- It operates on single 5 V power supply.
- Output meet TTL voltage level specifications.

Operation

When the \overline{WR} input goes low, the internal successive approximation register (SAR) is reset. As long as both \overline{CS} and \overline{WR} remain low, the analog to digital converter will remain in its reset state. One to eight clock periods after \overline{CS} or \overline{WR} makes a low-to-high transition, conversion starts. The \overline{INTR} signal is held high during conversion process. After conversion, \overline{INTR} goes low which is used as end of conversion signal. By making \overline{CS} and \overline{RD} signals low, an output can be read through DB_0 to DB_7 data signals.

Analog Inputs

As mentioned earlier, there are two analog inputs to ADC 0803/0804/0805 : $IN+$ and $IN-$. These inputs are connected to an internal operational amplifier and are differential inputs. The differential input rejects the common mode noise and eliminates offset at zero analog input voltage.

The Fig. 13.37 shows a few ways to use these differential inputs. The Fig 13.37 (a) shows the one way to use single input that can vary between 0 V and +5 V. Using another way variable voltage can be applied to the $V_{in}(-)$ pin so that the zero reference for $V_{in}(+)$ can be adjusted, as shown in Fig. 13.37 (b).

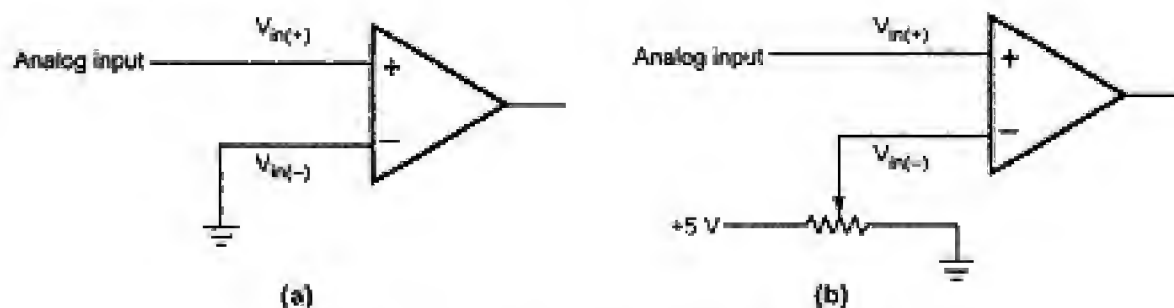


Fig. 13.37 Ways to use differential inputs

Clock Signal

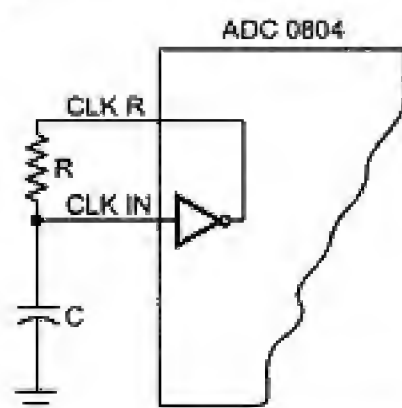


Fig. 13.38 Clock circuit

The ADC 0803/0804/0805 requires a clock source ranging 100 kHz to 1460 kHz for operation. The clock signal can be applied external or it can be generated with an RC circuit, as shown in the Fig. 13.38. When the RC circuit, shown in Fig. 13.38 is used to generate the clock, the clock frequency is given as,

$$F_{\text{clock}} = \frac{1}{1.1 RC}$$

Note : To have a minimum conversion time clock frequency should be kept nearer to 1460 kHz.

Typical Interface

Fig. 13.39 shows the interfacing of ADC 0803, ADC 0804 and ADC 0805 with microprocessor/microcontroller system.

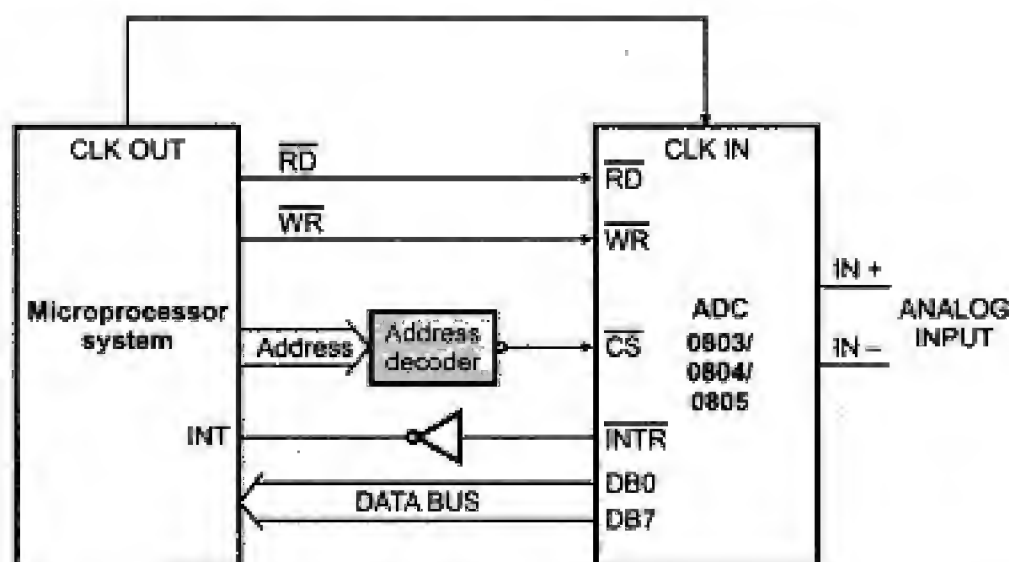


Fig. 13.39 Interfacing of ADC 0803/0804/0805 with microprocessor/microcontroller system

13.8.2 ADC 0808/0809 Family

The ADC 0808 and ADC 0809 are monolithic CMOS devices with an 8-channel multiplexer. These devices are also designed to operate from common microprocessor/microcontroller control buses, with tri-state output latches driving the data bus. The main features of these devices are :

Features

- 8-bit successive approximation ADC.
- 8-channel multiplexer with address logic.
- Conversion time 100 μ s.
- It eliminates the need for external zero and full-scale adjustments.
- Easy to interface to all microprocessors.
- It operates on single 5 V power supply.
- Output meet TTL voltage level specifications.

Pin Diagram

Fig. 13.40 shows pin diagram of 0808/0809 ADC.

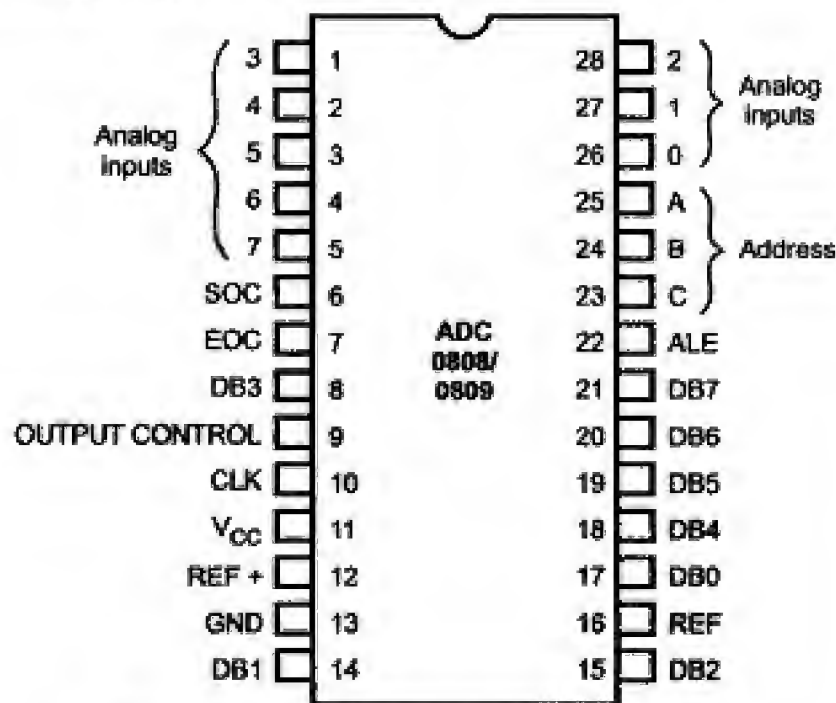


Fig. 13.40 Pin diagram of ADC 0808/0809

Operation

ADC 0808/0809 has eight input channels, so to select desired input channel, it is necessary to send 3-bit address on A, B and C inputs. The address of the desired channel is sent to the multiplexer address inputs through port pins. After atleast 50 ns, this address must be latched. This can be achieved by sending ALE signal. After another 2.5 μ s, the start of conversion (SOC) signal must be sent high and then low to start the conversion process. To indicate end of conversion ADC 0808/0809 activates EOC signal. The microprocessor system can read converted digital word through data bus by enabling the output enable signal after EOC is activated. This is illustrated in Fig. 13.41.

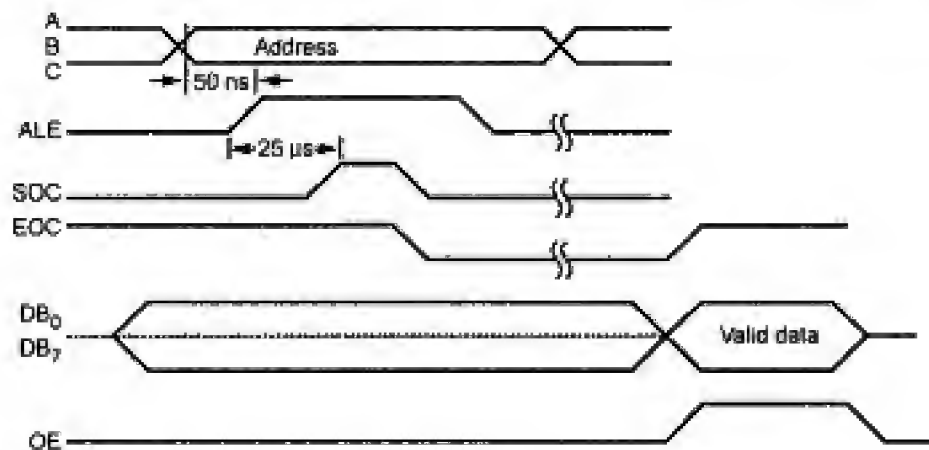


Fig. 13.41 Timing waveforms for ADC 0808

Interfacing

Fig. 13.42 shows typical interfacing circuit for ADC 0808 with microprocessor/microcontroller system.

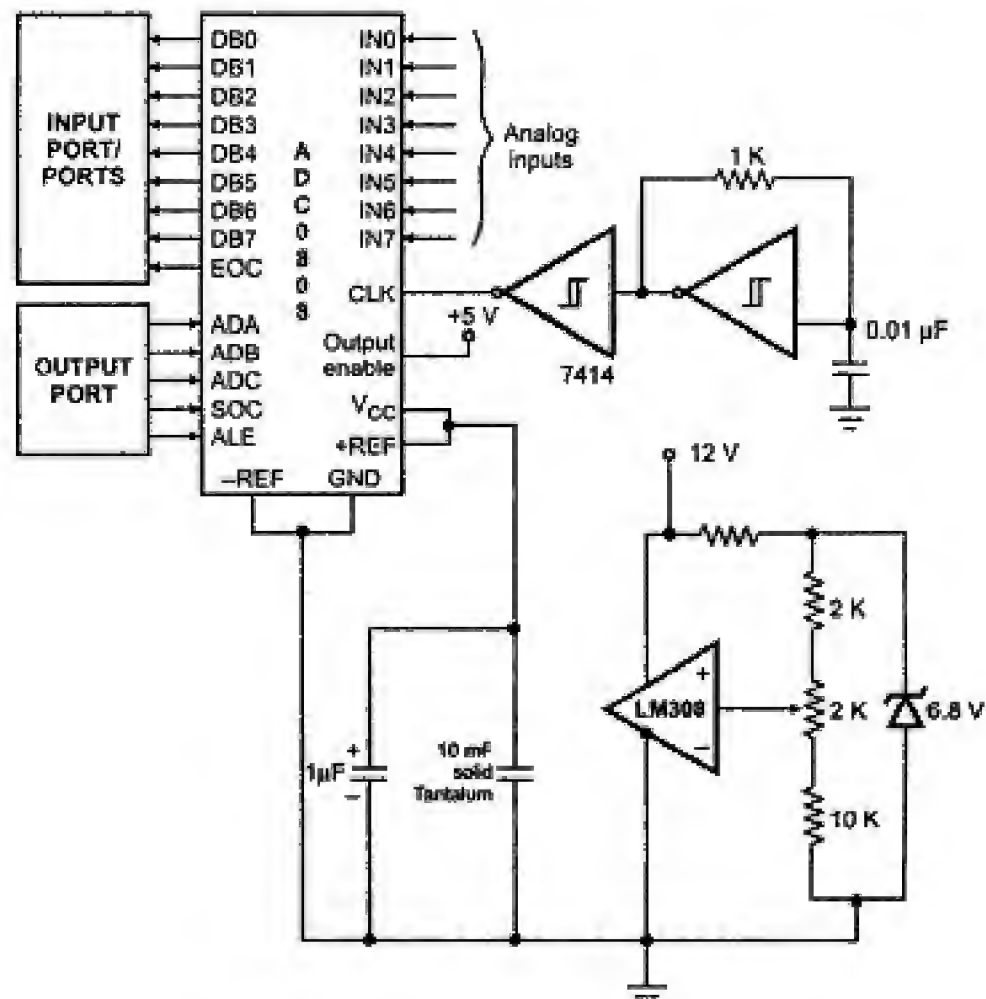


Fig. 13.42 Typical interface for 0808/0809

The zener diode and LM 308 amplifier circuitry is used to produce a V_{CC} and V_{REF} of 5.12 V for the A/D converter. With this reference voltage the A/D converter will have 256 steps of 20 mV each.

➔ **Example 13.3 :** Interface 8 bit, 8 channel ADC to 8051. Write assembly language program to convert CH0, CH3 and CH7 and store result in external memory location starting from C000H. Repeat procedure for every 1 sec.

Solution : The Fig. 13.43 shows the interfacing of 8 channel, 8-bit ADC to 8051.

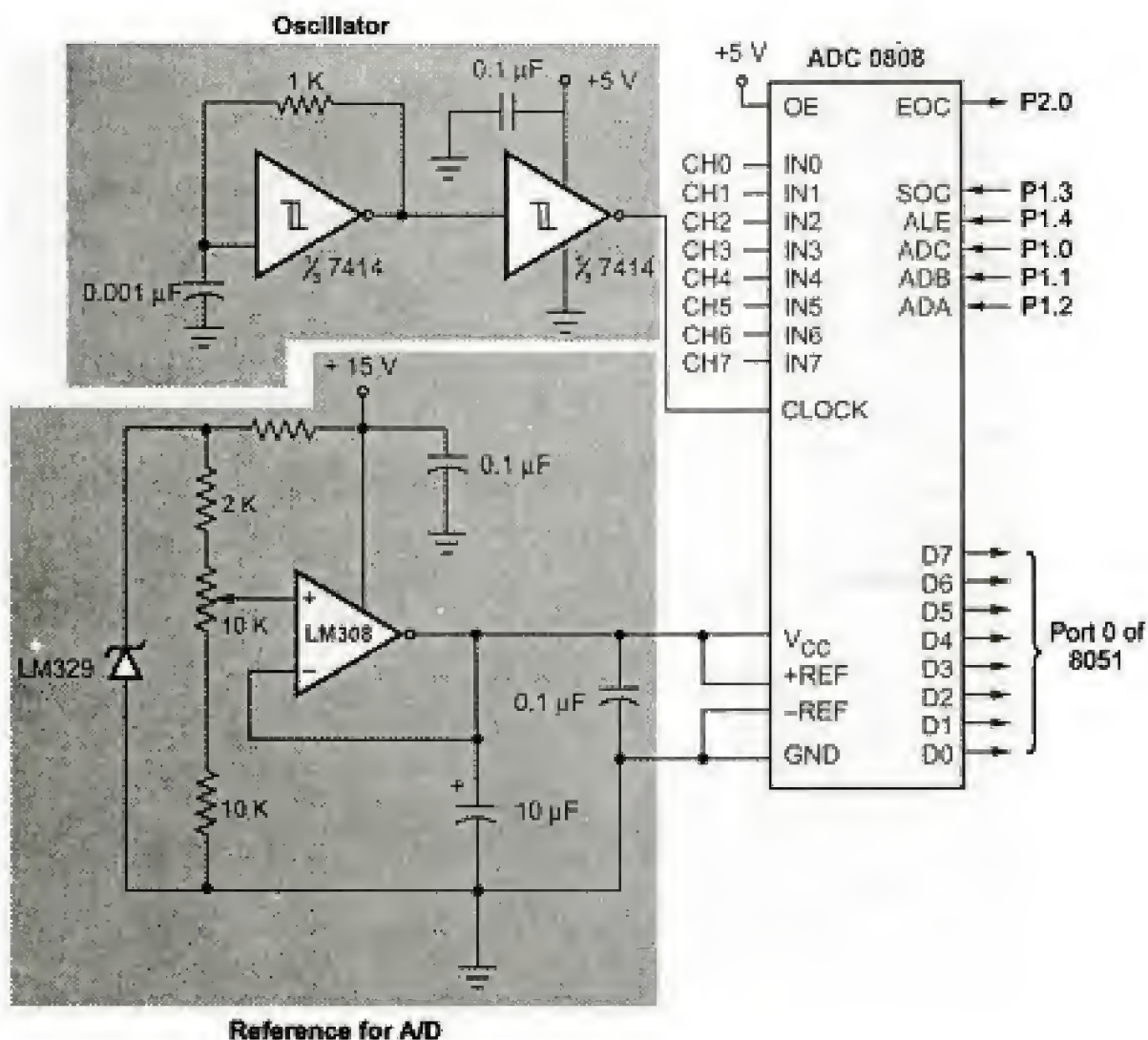


Fig. 13.43

ADC 0808 has eight input channels, so to select desired input channel, it is necessary to send 3-bit address on ADC, ADB, ADA inputs. The address of the desired channel is sent to the address inputs through port pin P1.0, P1.1 and P1.2 of 8051.

After at least 50 ns, this address must be latched. This can be achieved by sending ALE signal. After another 2.5 μ s, the start of conversion (SOC) signal must be sent high and then low to start the conversion process. To indicate end of conversion (EOC) (ADC 0808/0809) activates EOC signal. The 8051 pins P2.0 and P1.3 are connected to EOC and SOC pin of 0808/0809.

After conversion is over, 8-bit digital data is available on D₀ through D₇ lines. The 8051 accepts this data through port 0.

Program

```

CLR P1.3          ; Make SOC low
CLR P1.4          ; Make ALE low
MOV P0, #0FFH     ; Configure port 0 as input
MOV P2, #0FFH     ; Configure port 2 as input
BACK : MOV DPTR, #0C000H ; Initialize memory pointer
      MOV A, #00H   ; Set address for channel 0
      ACALL R_ADC   ; Call ADC Routine
      MOVX @DPTR, A ; Increment memory pointer
      MOV A, #03H   ; Set address for channel 3
      ACALL R_ADC   ; Call ADC Routine
      MOVX @DPTR, A ; Save digital value
      INC DPTR      ; Increment memory pointer
      MOV A, #07H   ; Set address for channel 7
      ACALL R_ADC   ; Call ADC Routine
      MOVX @DPTR, A ; Save digital value
      ACALL DELAY   ; Wait for 1 sec
      SJMP BACK     ; Repeat

```

Analog to Digital Conversion Routine

```

R_ADC : MOV P1, A   ; Get the channel number and set
                  ; its address
      SET P1.4      ; Send ALE
      NOP           ; Pulse
      CLR P1.4
      SET P1.3      ; Send start of
      NOP           ; Conversion
      CLR P1.3      ; Pulse
WAIT : JB P2.0, WAIT ; Wait for EOC signal
      MOV A, P0     ; Get digital data
      RET          ; Return

```

Delay Routine

```

DELAY : MOV TMOD, #01 ; Timer 0, mode 1 (16-bit mode)
      MOV R0, #14H    ; Initialize counter to 20
BACK : MOV TL0, #B0H  ; TL0 = B0H, the low Byte
      MOV TH0, #3CH   ; TH0 = 3CH, the high byte
      SETB TR0        ; Start the timer 0
AGAIN : JNB TF0, AGAIN ; Check timer0 flag until
                  ; it rolls over
      CLR TR0         ; Stop timer 0

```

```

CLR TF0          ; Clear timer 0 flag
DJNZ R0, BACK    ; Decrement counter and if
                  ; not zero repeat
RET              ; Return

```

Note : Timer 0 gives a delay of 50 ms. To get a delay of 1 sec, such delay is executed 20 times.

13.8.3 Interfacing of ADC 0803/0804/0805 with 8051

The Fig. 13.44 shows the interfacing of ADC 0803/0804/0805 with 8051 using port 1 and port 2. Here, port 1 is used to read digital data from ADC and port 2 is used to provide control signals to ADC 0803/0804/0805. Potentiometer is used to adjust V_{in+} voltage. The clock signal is provided using internal clock generator and two external components, resistor and capacitor as shown in the Fig. 13.44. The frequency of such clock can be determined by,

$$f = \frac{1}{11RC}$$

The typical values are $R = 10\text{ k}\Omega$ and $C = 150\text{ pF}$. With these values we get approximately 606 kHz clock frequency. In such case, the conversion time is around $110\text{ }\mu\text{s}$.

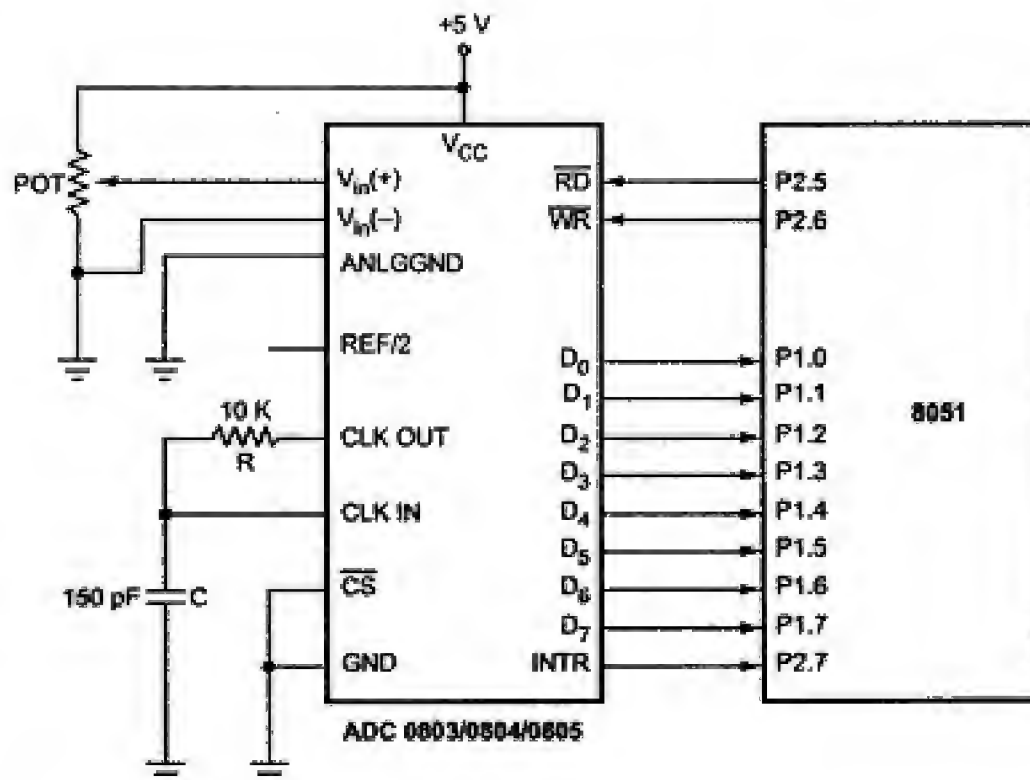


Fig. 13.44

AD Conversion Program

```

MOV P1, #0FFH    ; configure Port1 as input
BACK: CLR P2.6    ; [Make WR = 0 and

```

```

        SETB P2.6           ; Make  $\overline{WR} = 1$  to generate start of
                              ; conversion pulse]
AGAIN:   JB P2.7, AGAIN     ; Wait for end of conversion
        CLR P2.5           ; Enable read
        MOV A, P1          ; Read data through port1
        SETB P2.5         ; Disable read after reading data
        SJMP BACK          ; go for next conversion cycle

```

Practical Applications

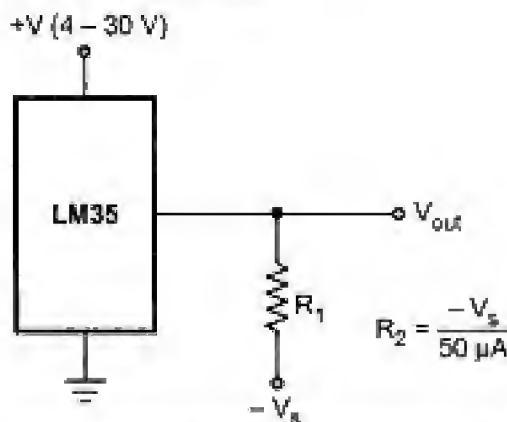


Fig. 13.45 Temperature dependent voltage source using LM35

The National LM35 is a temperature sensor. It is a temperature-sensitive voltage source. Its output voltage increases by 10 mV for each $^{\circ}\text{C}$ increase in its temperature. The Fig. 13.45 shows the circuit connection for temperature sensor LM35. The voltage output from this circuit is connected to a negative reference voltage, V_s as shown, the sensor will give a meaningful output for a temperature range of -55 to 150°C . The output is adjusted to 0 V for 0°C . The output voltage can be amplified to give the voltage range you need for a particular

application. The output voltage from LM35 can be applied to ADC and we can get digital equivalent of analog voltage corresponding to current temperature.

The AD590 is another commonly used temperature sensor. It is temperature sensitive current source. It produces a current of $1 \mu\text{A}/^{\circ}\text{K}$. This current can be converted to voltage source by current to voltage converter. The advantage of current source sensor is that voltage drop in long connecting wires do not have any effect on the output value. The output of amplifier shown in Fig. 13.46 can be applied to ADC to get the digital equivalent of current temperature.

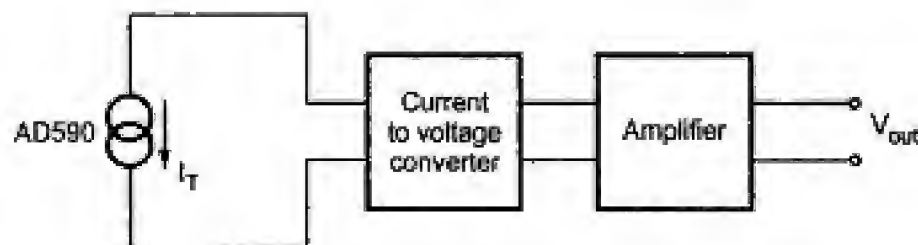


Fig. 13.46 Temperature sensor using AD590

13.9 Stepper Motor Interface

A stepper motor is a digital motor. It can be driven by digital signal. Fig. 13.47 shows the typical 2 phase motor interfaced using 8255. Motor shown in the circuit has two phases, with center-tap winding. The center taps of these windings are connected to the

12 V supply. Due to this, motor can be excited by grounding four terminals of the two windings. Motor can be rotated in steps by giving proper excitation sequence to these windings. The lower nibble of port A of the 8255 is used to generate excitation signals in the proper sequence.

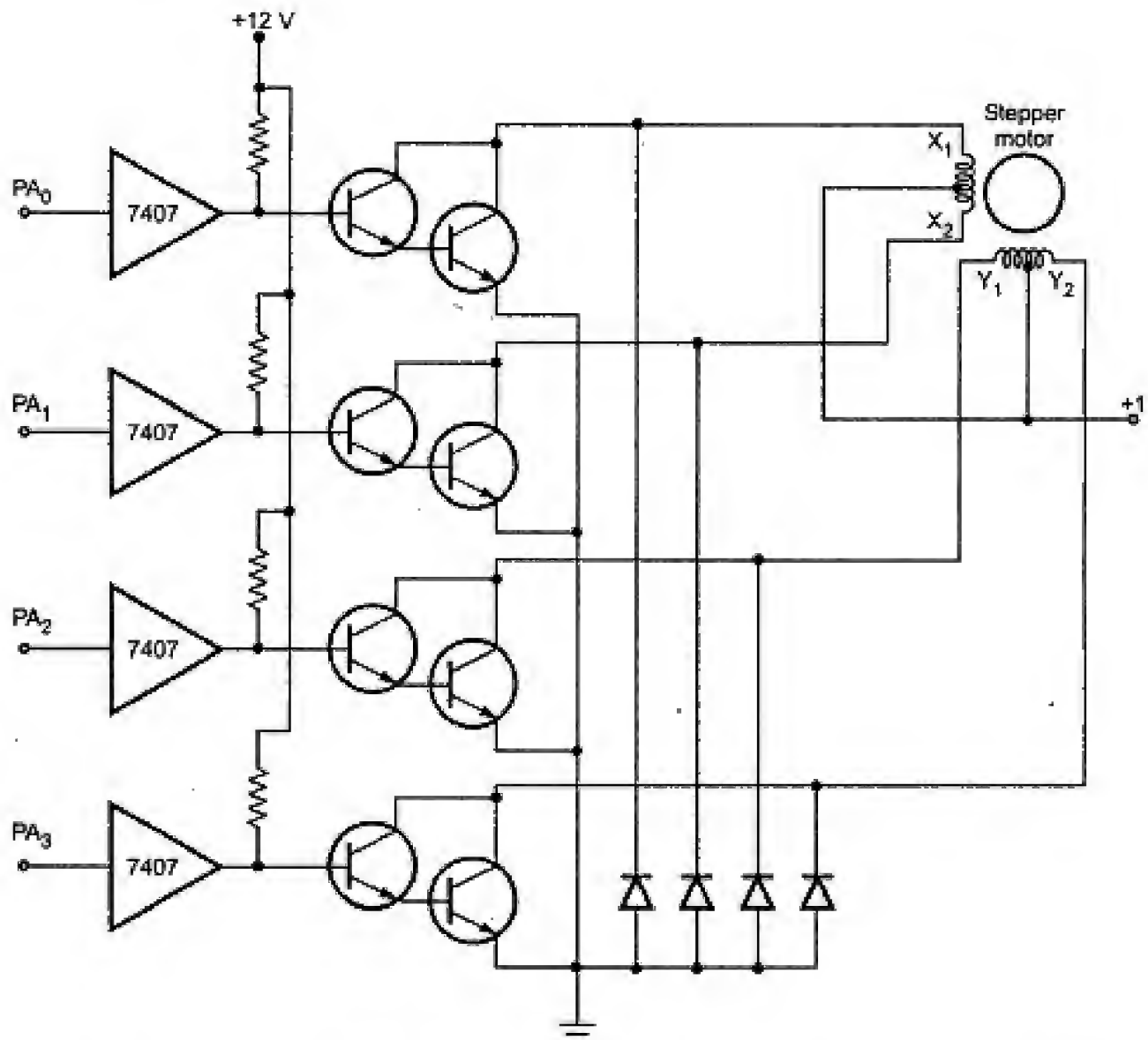


Fig. 13.47 Stepper motor interface

The Table 13.12 shows typical excitation sequence. The given excitation sequence rotates the motor in clockwise direction. To rotate motor in anticlockwise direction we have to excite motor in a reverse sequence. The excitation sequence for stepper motor may change due to change in winding connections. However, it is not desirable to excite both the ends of the same winding simultaneously. This cancels the flux and motor winding may damage. To avoid this, digital locking system must be designed. Fig. 13.48

shows a simple digital locking system. Only one output is activated (made low) when properly excited; otherwise output is disabled (made high).

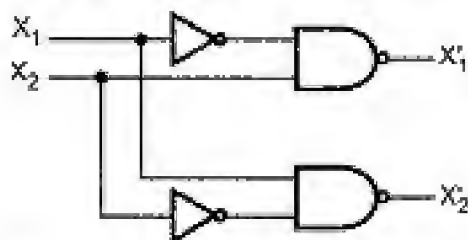


Fig. 13.48 Digital locking system

| Step | X_1 | X_2 | Y_1 | Y_2 |
|------|-------|-------|-------|-------|
| 1 | 0 | 1 | 0 | 1 |
| 2 | 1 | 0 | 0 | 1 |
| 3 | 1 | 0 | 1 | 0 |
| 4 | 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |

Table 13.12 Full step excitation sequence

The excitation sequence given in Table 13.12 is called full step sequence. In which excitation ends of the phase are changed in one step. The excitation sequence given in Table 13.13 takes two steps to change the excitation ends of the phase. Such a sequence is called half step sequence and in each step the motor is rotated by 0.9° .

| Step | X_1 | X_2 | Y_1 | Y_2 |
|------|-------|-------|-------|-------|
| 1 | 0 | 1 | 0 | 1 |
| 2 | 0 | 0 | 0 | 1 |
| 3 | 1 | 0 | 0 | 1 |
| 4 | 1 | 0 | 0 | 0 |
| 5 | 1 | 0 | 1 | 0 |
| 6 | 0 | 0 | 1 | 0 |
| 7 | 0 | 1 | 1 | 0 |
| 8 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 0 | 1 |

Table 13.13 Half step excitation sequence

We know that stepper motor is stepped from one position to the next by changing the currents through the fields in the motor. The winding inductance opposes the change in current and this puts limit on the stepping rate. For higher stepping rates and more

torque, it is necessary to use a higher voltage source and current limiting resistors as shown in Fig. 13.49. By adding series resistance, we decrease L/R time constant, which allows the current to change more rapidly in the windings. There is a power loss across series resistor, but designer has to compromise between power and speed.

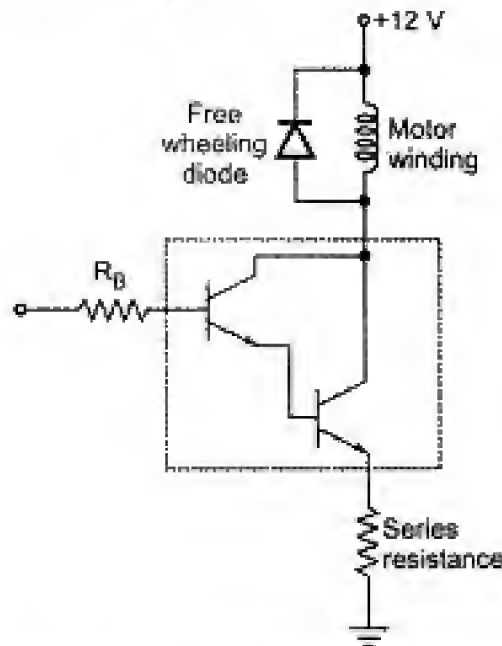


Fig. 13.49 Excitation circuit with series resistance

➡ **Example 13.4 :** Write an 8051 assembly language program to control stepper motor using connections given in Fig. 13.47.

Solution :

```

MOV R0, # COUNT      ; Initialize rotation count
AGAIN : MOV DPTR, #ETC ; Initialize pointer to
                   ; excitation code table
MOV R1, #04           ; Initialize counter to excitation
                   ; code sequence

BACK : MOVX A, @DPTR   ; Get the excitation code
MOV P1, A             ; send the excitation code
LCALL DELAY           ; Wait for some time
INC DPTR              ; Increment pointer
DJNZ R1, BACK         ; Decrement R1; if not zero goto BACK
DJNZ R0, AGAIN        ; Decrement R0; if not zero goto AGAIN
RET
ORG 3000H
ETC DB 03H, 06H, 09H, 0CH; code sequence for clockwise rotation

```

➡ **Example 13.5 :** Write assembly language program to control conveyor belt using stepper motor and 8051 controller. Belt moves continuously at rate of 1 step/sec. but stops for 5 sec. when external interrupt occurs and then continues to move.

Solution :

```

MAIN:    MOV IE, #1000 0001B    ; Enable external interrupt 0
AGAIN:   MOV DPTR, #ETC         ; Initialize pointer to
                                ; excitation code table
                                ;
                                ; Initialize counter to excitation
                                ; code sequence
BACK:    MOVX A, @DPTR          ; Get the excitation code
        MOV P1, A              ; send the excitation code
        MOV A, #14H            ; Initialize count = 20
        LCALL DELAY            ; Wait for 1sec
        INC DPTR               ; Increment pointer
        DJNZ R1, BACK          ; Decrement R1
                                ; if not zero goto BACK
        SJMP AGAIN             ; Repeat
        ORG 3000H
        ETC DB 03H, 06H, 09H, 0CH ; Code sequence for
                                ; Clockwise rotation

```

Assume external interrupt INT0 is used.

```

        ORG 0003H
        MOV A, #64H            ; Initialize count = 100
        ACALL DELAY            ; Called delay routine
        RETI                   ; Return to main program

```

Delay Routine

```

DELAY:   MOV TMOD, #01         ; Time 0, mode1 (16-bit mode)
        MOV R0, A              ; Read count and initialize
BACK:    MOV TL0, #B0H         ; TL0=B0H, the low byte
        MOV TH0, #3CH         ; TH0=3CH, the high byte
        SETB TR0              ; Start the timer 0
RERE:    JNB TF0, REPE         ; Check timer 0 flag until
                                ; it rolls over
        CLR TR0               ; Stop timer 0
        CLR TF0               ; clear timer 0 flag
        DJNZ R0, BACK          ; Decrement counter and
        RET                   ; if not zero repeat

```

Note : Timer 0 gives a delay of 50 ms.

Therefore, to get delay of 1 sec = 50 ms × 20

We load 20 as a count and to get delay of 5 sec. = 50 ms × 100

we load 100 as count.

13.10 Typical MCS-51 Based System

Fig. 13.50 shows a typical MCS-51 based system, which includes a port expander, 8K EPROM, 8K RAM.

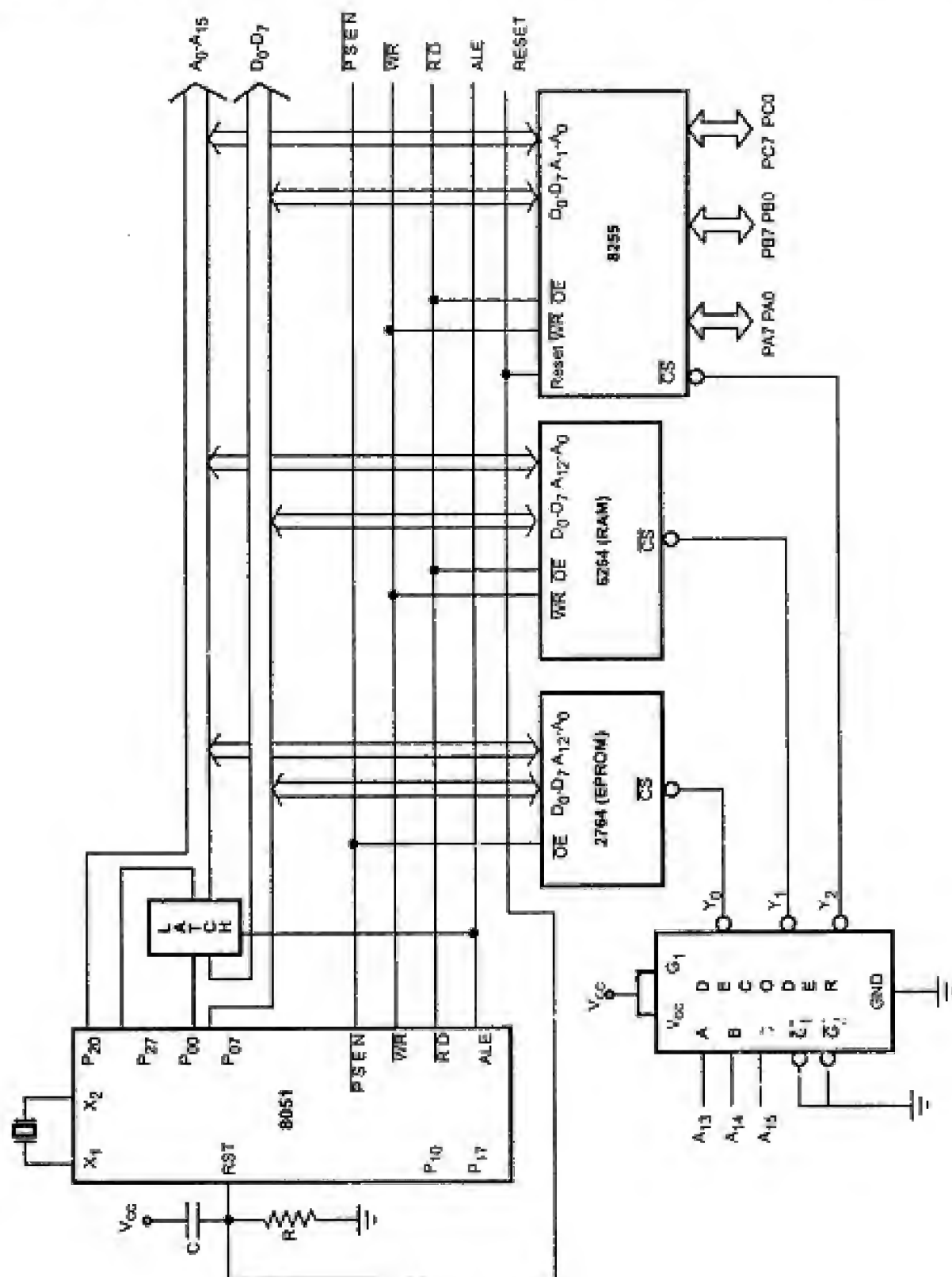


Fig. 13.50

13.11 Interfacing Examples

➡ **Example 13.6 :** Implement programmable frequency divider using timer of 8051.

Solution : The Fig. 13.51 shows the connection for timer operation. The internal timer 0 is used to implement programmable frequency divider. The DIP switch is used to read the divider count. The following section gives the algorithm and program required for this application.

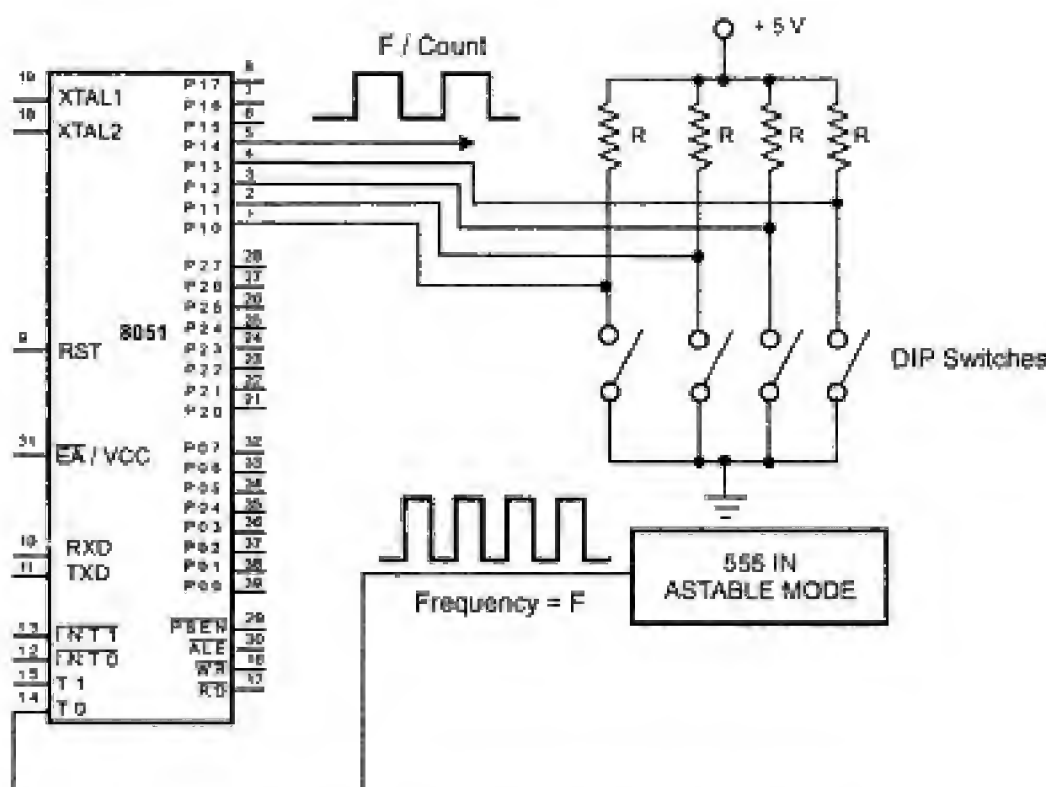


Fig. 13.51 Frequency divider application

Algorithm :

1. Initialise port pins P1.0, P1.1, P1.2, P1.3 as input pins by writing '1' s to it.
2. Disable all the interrupt. Clear line P1.4 to show low level.
3. Read the binary count = say 'n' (n = 2, 4, 6 etc.)
4. Initialise timer 0 as counter in mode '1', disable counter by clearing TCON regi (i.e. clearing TR0 bit)
5. Divide the binary count by 2 by shift right operation.
6. Take the complement of the (count/2) number and add 1 to it giving 2's complement of the (count/2) number.
7. Start the counter operation by setting TR0 bit.

8. Wait for TF0 bit to set. (This bit indicates timer overflow).
9. After the overflow of the counter, clear TR0 i.e. stop the counter. Clear TF0 (reset timer overflow bit). Complement P1.4.
10. Go to step 3 to check if the divide count has been changed.

Program :

```

org program_address
mov sp, #address_of_stack_area
mov pl, #0ffh
clr a
mov ie, a          ; clear int
mov tmod, a        ; initialise timer mode
mov tcon, a        ;
setb p3.4          ;

```

| : | GATE | C/T | M1 | M0 | GATE | C/T | M1 | M0 |
|---|------|-----|----|----|------|-----|----|----|
| : | x | x | x | x | 0 | 1 | 0 | 1 |

Timer 1**Timer 0**

```

back:   mov tmod, #05          ; mode 1, timer 0, counter
        mov th0, #0ffh
        clr pl.0              ; make common to grid
        mov a, pl
        anl a, #00011110b     ; just take 4 bits
        rr a                  ; adjust proper position
        cpl a                 ; take compliment
        anl a, #0fh           ; get actual BCD value from
                                ; DIP switch
        rr a                   ; divide by two
        cpl a                 ;
        inc a                  ; take 2's complement of a number
        mov tl0, a            ; load into low timer
        setb tr0              ; enable timer
HERE :   jnb tf0, HERE
        clr tr0               ; disable timer
        cpl pl.5
        clr tf0               ; reset tf0
        ajmp back
        end

```


➡ **Example 13.7 :** Design microcontroller system to control traffic lights. The traffic light arrangement is as shown in Fig. 13.52. The traffic should be controlled in the following manner.

- 1) Allow traffic from W to E and E to W transition for 20 seconds.
- 2) Give transition period of 5 seconds (Yellow bulbs ON).
- 3) Allow traffic from N to S and S to N for 20 seconds.
- 4) Give transition period of 5 seconds (Yellow bulbs ON).
- 5) Repeat the process.

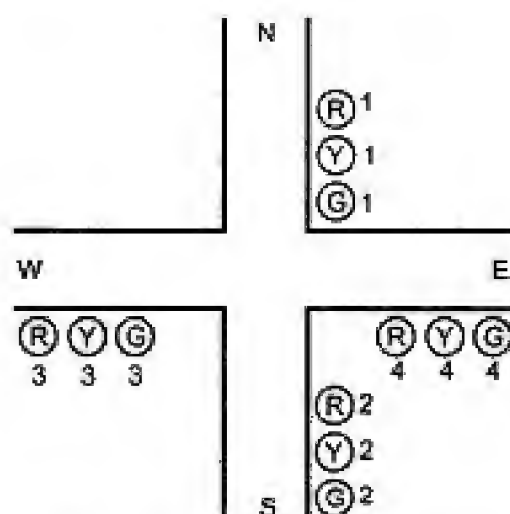


Fig. 13.52 Traffic light control

Solution :

Hardware : Fig. 13.53 shows the interfacing diagram to control 12 electric bulbs. Port 1 pins are used to control lights as shown in Table 13.14.

| Pin | Control |
|-----------|----------------------|
| Port P1.0 | R_1, R_2, G_3, G_4 |
| Port P1.1 | Y_1, Y_2, Y_3, Y_4 |
| Port P1.2 | R_3, R_4, G_1, G_2 |

Table 13.14

Electric bulbs are controlled by relays. The port 1 pins are used to control relay ON-OFF action with the relay driver circuit. The driver circuit includes 3 transistors to drive 3 relays.

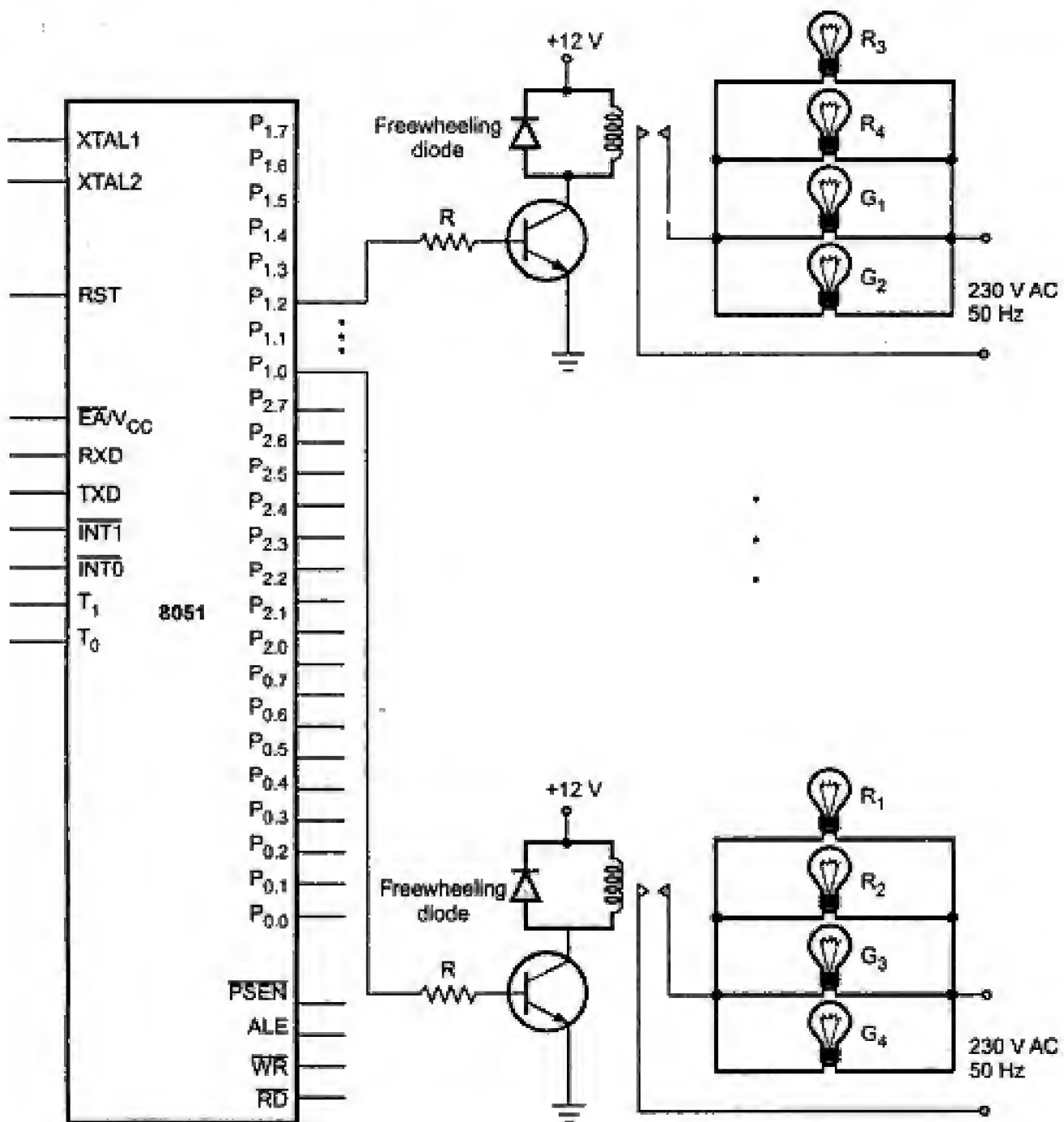


Fig. 13.53 Interfacing diagram for traffic light control system

Program :

```

MOV r0, # 01H      ; Initialize r0 to make P1.0 high
MOV r1, # 02H      ; Initialize r1 to make P1.1 high
MOV r2, # 04H      ; Initialize r2 to make P1.2 high
START: MOV P1, r0H  ; Make P1.0 high
        LCALL Delay1 ; Wait for 20 seconds
        MOV P1, r1  ; Make P1.1 high
        LCALL Delay2 ; Wait for 5 seconds

```

```

MOV P1, r2      ; Make P1.2 high
LCALL Delay1    ; Wait for 20 seconds
MOV P1, r1      ; Make P1.1 high
LCALL Delay2    ; Wait for 5 seconds
LJMP START      ; Jump to start

```

➡ **Example 13.8 :** Part 1 of 8051 is to be connected to two on-off switches and two LED. It is required to sense the status of the switches and indicate it through the LEDs. Write a program to accomplish this task and also give the necessary interfacing details.

Solution : Fig. 13.54 shows the interfacing diagram. P1.0 and P1.1 are used to connect LEDs while P1.2 and P1.3 are used to connect switches. The status of the switches are sensed by BIT TEST instruction and LEDs are driven by BIT SET instructions.

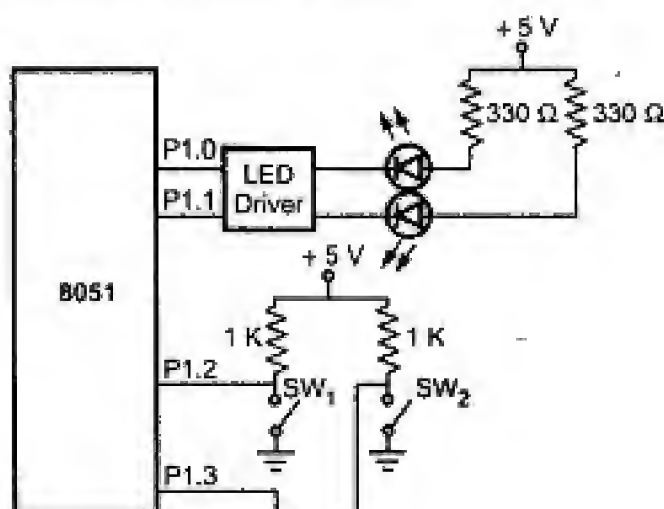


Fig. 13.54

Program :

```

MOV P1, # 0Ch      ; configure P1.2 and P1.3
                   ; lines of part 1 as input

BACK      :      JNB P1.2,GLOW_L1
                SETB P1.0
                JNB P1.3,GLOW_L2
                SETB P1.1
                JMP BACK      ; keep polling

GLOW_L1   :      CLR P1.0
                JMP BACK      ; keep polling

GLOW_L2   :      CLR P1.1
                JMP BACK      ; keep polling

```

Note : By clearing bit LED will glow.

➡ **Example 13.9 :** Draw a block schematic of following system configuration for 8051, 8 K EPROM, 32 K RAM, 8255. Indicate addresses generated through the chip select logic for above devices.

Solution : Fig. 13.55 shows the block schematic of system configuration.

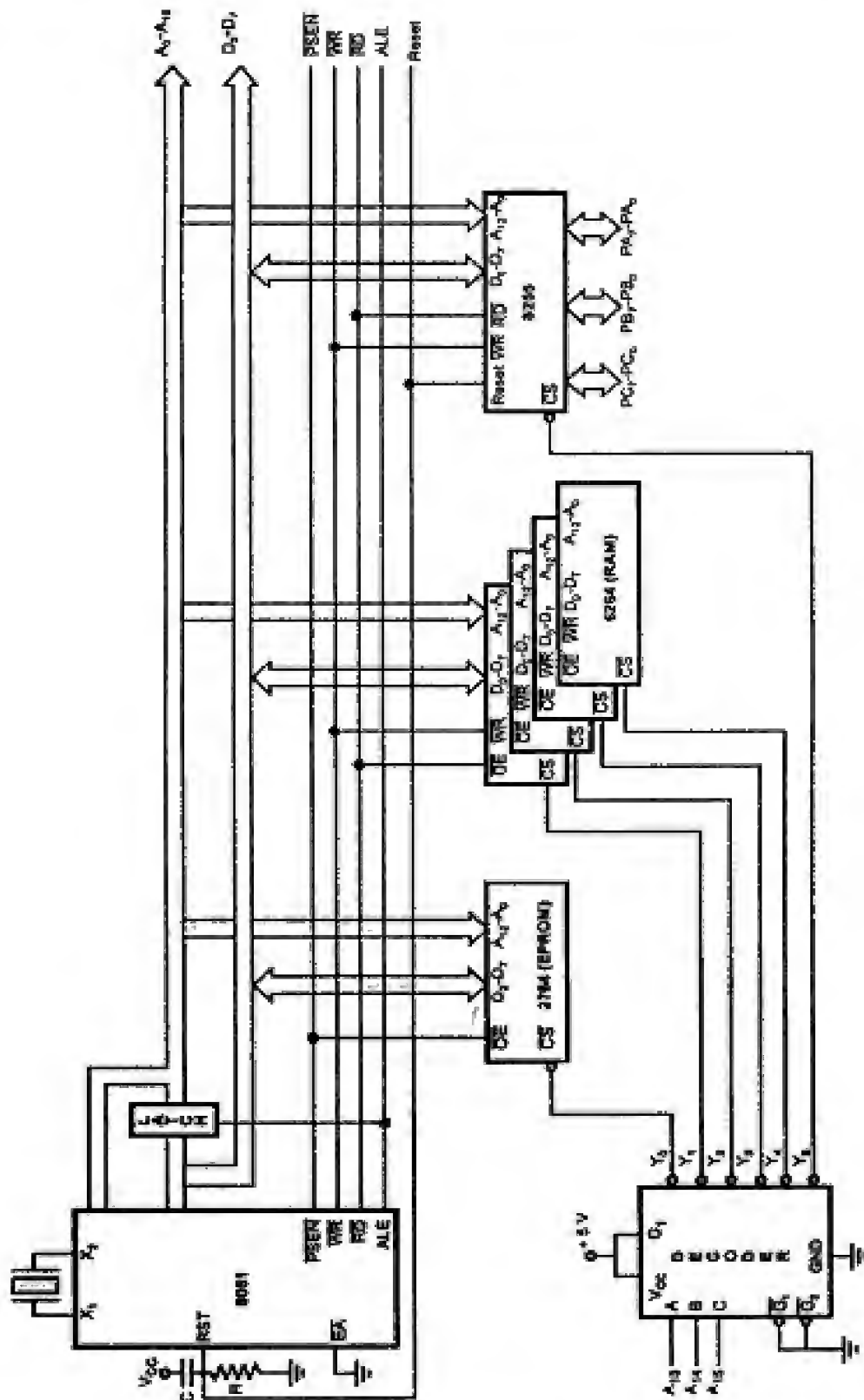


Fig. 13.55

Address Map :

| | |
|------------------------|-------|
| EPROM Starting Address | 0000H |
| EPROM End Address | 1FFFH |
| RAM Starting Address | 2000H |
| RAM End Address | 9FFFH |
| 8255 Port A | 8000H |
| 8255 Port B | 8001H |
| 8255 Port C | 8002H |
| 8255 Control Register | 8003H |

Table 13.15

➡ **Example 13.10 :** Write an 8051 assembly language program to flash alternate LEDs connected to port B of 8255, with 1 sec. delay.

Solution :

Assume : PORT A address = 8000H
 PORT B address = 8001H
 PORT C address = 8002H
 CR address = 8003H

```

MOV A, #80H      ; Load control word
MOV DPTR, #8003H ; Initialize pointer to point CR
MOVX @DPTR, A    ; Send control word to CR
MOV A, #55H      ; Load alternate pattern
MOV DPTR, #8001H ; Initialize pointer to point port B
BACK : MOVX @DPTR, A ; Send alternate pattern to PORT B
      LCALL DELAY   ; Wait for 1 sec
      CPL A         ; Complement A
      SJMP BACK     ; Repeat

```

Refer Delay Routine given in example 9.8.

➡ **Example 13.11 :** Write an assembly language program to flash PC_0 and PC_2 alternately with delay of 10 ms. Show the delay calculations.

Solution :

Assume CR address of 8255 = 8003H

```

      MOV DPTR, #8003H ; Initialize pointer to CR
BACK : MOVX A, #00H    ; Make  $PC_0 = 0$ 
      MOVX @DPTR, A
      MOV A, #05H      ; Make  $PC_2 = 1$ 

```

```

MOVX @DPTR, A
LCALL DELAY      ; Wait for 10 ms
MOV A, #01H      ; Make PC0 = 1
MOVX @DPTR, A
MOVA, #04H       ; Make PC2 = 0
MOVX @DPTR, A
LCALL DELAY      ; Wait for 10 ms
SJMP BACK

```

Delay Routine

We use timer 0 to generate a delay of 10 ms. Assume XTAL = 12 MHz $T = 1 \mu s$.

To generate a delay of 10 ms the count to be loaded in TH0 and TL0 is given by,

$$\text{Count} = 65536 - 10000 = (55536)_{10} = (D8F0)H$$

```

DELAY:  MOV TMOD, #01      ; Timer 0, mode 1 (16-bit mode)
        MOV TL0, #0F0H     ; TL0 = 0F0H, the low byte
        MOV TH0, #008H     ; TH0 = 008H, the high byte
        SETB TR0           ; Start the timer 0
AGAIN:  JNB TF0, AGAIN     ; Check timer 0 flag until
                        ; its rolls over
        CLR TR0            ; Stop timer 0
        CLR TF0            ; Clear timer 0 flag
        RET                ; Return

```

➡ **Example 13.12 :** Design a 8051 based microcontroller system with following details :-

- 8051 CPU at 10 MHz.
- Program memory 16 Kbytes.
- Data memory 32 Kbytes.
- 4 seven-segment LED display.

Discuss the designed system with diagram.

Solution : Fig. 13.56 shows 8051 based microcontroller system.

For interfacing memory to the 8051 Port 0 and Port 2 are used as multiplexed address/data bus and a higher order address bus respectively. External latch is used to demultiplexed address and data. As per the specifications 16 K EPROM and 32 K RAM chips are connected to the system bus.

External decoder is used to generate chip select signals for memory chips as well as for 8279, keyboard/display controller. The 8279 is used in decoded mode to interface four 7-segment digits. Its B_0-B_3 and A_0-A_3 lines give data to be displayed and S_0-S_3 lines select the proper 7-segment digit.

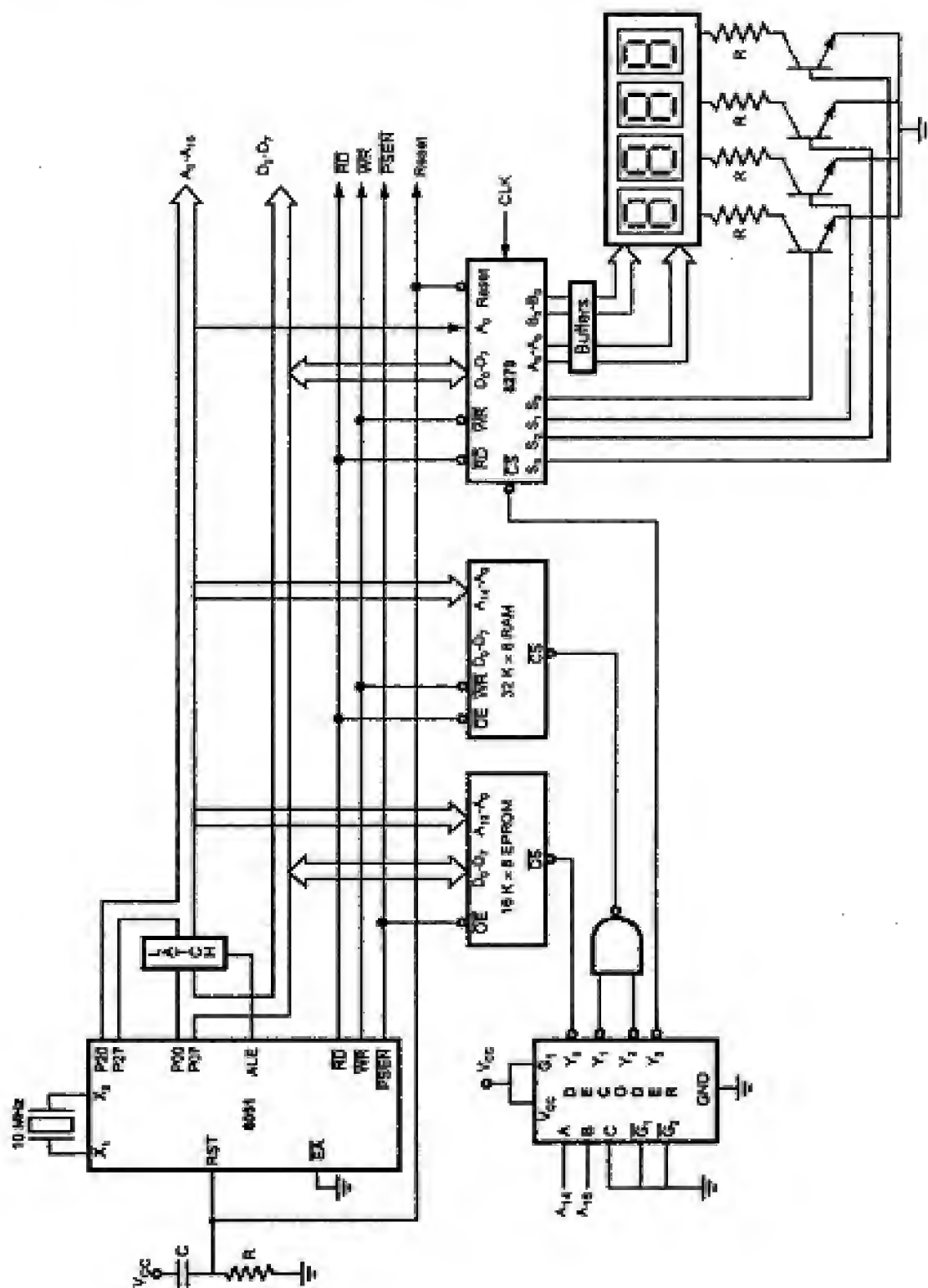


Fig. 13.56

Review Questions

1. Explain the external program memory interface of 8051.
2. Explain the external data memory interface of 8051.
3. List the instructions to access external ROM.
4. List the instructions to access external RAM.
5. Discuss important points in accessing external memory.
6. How to achieve I/O expansion in 8051 ?
7. What is keybounce ? How it is achieved ?
8. Explain the matrix keyboard interface of 8051.
9. Interface 8×4 key matrix keyboard to the 8051.
10. What do you mean by key debounce?
11. What do you mean by static display and multiplexed display.
12. Interface 4 digit 7-segment LED display to 8051 and write an assembly language program to display 1234 on it.
13. Write a short note on LCD displays.
14. Draw and explain the interfacing 4 digit nonmultiplexed 7-segment LCD display.
15. Write a note on BCD to 7-segment LCD decoder/driver.
16. Draw and explain the interfacing of LCD module to 8051.
17. Explain various commands associated with LCD module.
18. Write an assembly language program to display any message on the LCD module.
19. Explain 1408 DAC with the help of block diagram.
20. Draw and explain the typical interfacing circuit for DAC 1408.
21. Draw and explain the typical interfacing circuit for DAC 1408 in the bipolar range.
22. Give the important electrical characteristics for 1408 DAC.
23. Give a complete scheme to interface an 8-bit ADC to 8051 microcontroller. Draw flowchart and write an assembly language program.
24. List the features of ADC 0804.
25. Explain the functions of various pins of ADC 0804.
26. Draw and explain the interfacing of ADC 0804 to 8051.
27. Draw and explain the stepper motor interface.
28. Explain the methods for excitation of stepper motor.
29. Explain the limitation for increasing the speed of stepper motor.
30. Interface stepper motor to the 8051 microcontroller and write an assembly language program to rotate it 180° .



Buses and Protocols

14.1 Introduction

The processor, main memory and I/O devices can be interconnected by means of a common bus. A bus is a set of lines (wires) designed to transfer all bits of a word from a specified source to a specified destination. Thus bus provides a communication path for the transfer of data. The bus includes data lines, address lines and control lines. It also includes the lines needed to support interrupts and arbitration as a part of control lines. In this section, we discuss the main features of the bus protocols used for transferring data. Protocol defines certain set of rules that govern the behaviour of various devices connected to the bus. It defines when to place information on the bus, when to assert control signals and so on.

When processor, memory and I/O devices are connected in the system one particular unit acts as the bus master and supervises the use of the bus by the other units, the bus slaves. In most of the cases the processor is the bus master, while the memory and I/O interface circuits are slaves. However, I/O controllers also serve as bus masters. Only a bus master can initiate data transfer, although slaves can request for data transfer. Once the data transfer is initiated, both master and slave participate equally in the data transfer process.

Most of the microprocessors/microcontroller are designed for parallel communication. In parallel communication number of lines required to transfer data depend on the number of bits to be transferred. For example, to transfer a byte of data, 8 lines are required and all 8 bits are transferred simultaneously. Thus for transmitting data over a long distance, using parallel communication is impractical due to the increase in cost of cabling. Parallel communication is also not practical for devices such as cassette tapes or a CRT terminal. In such situations, serial communication is used. In serial communication one bit is transferred at a time over a single line.

In this chapter, we are going to see the classification of serial data communication and various protocols by which we can carry out serial communication.

14.2 Classification of Serial Data Communication

Serial data transmission can be classified on the basis of how transmission occurs.

1. Simplex
2. Half duplex
3. Full duplex

14.2.1 Simplex

In simplex, the hardware exists such that data transfer takes place only in one direction. There is no possibility of data transfer in the other direction. A typical example is transmission from a computer to the printer.

14.2.2 Half Duplex

The half duplex transmission allows the data transfer in both directions, but not simultaneously. A typical example is a walkie-talkie.

14.2.3 Full Duplex

The full duplex transmission allows the data transfer in both direction simultaneously. The typical example is transmission through telephone lines.

14.3 Basic Serial Communication Data Formats

The data in the serial communication may be sent in two formats :

- a) Asynchronous
- b) Synchronous

14.3.1 Asynchronous

Fig. 14.1 shows the transmission format for asynchronous transmission. Asynchronous formats are character oriented. In this, the bits of a character or data word are sent at a constant rate, but characters can come at any rate (asynchronously) as long as they do not overlap. When no characters are being sent, a line stays high at logic 1 called mark, logic 0 is called space. The beginning of a character is indicated by a start bit which is always low. This is used to synchronize the transmitter and receiver. After the start bit, the data bits are sent with least significant bit first, followed by one or more stop bits (active high). The stop bits indicate the end of character. Different systems use 1, 1 1/2 or 2 stop bits. The combination of start bit, character and stop bits is known as frame. The start and stop bits carry no information, but are required because of the asynchronous nature of data. Fig. 14.2 illustrates how the data byte CAH would look when transmitted in the asynchronous serial format.

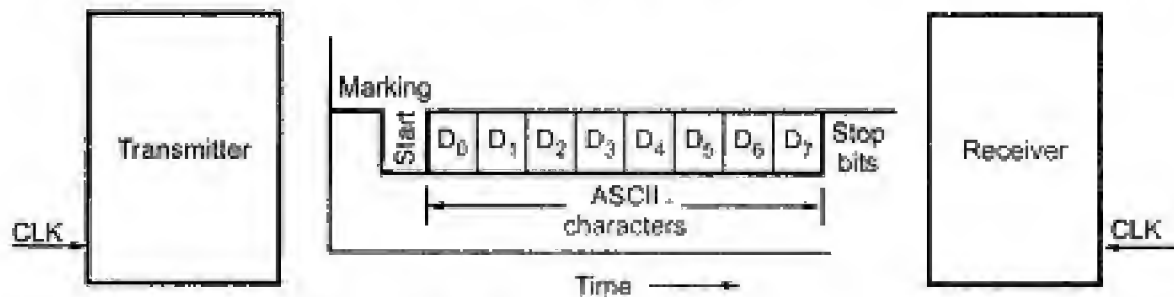


Fig. 14.1 Transmission format for asynchronous transmission

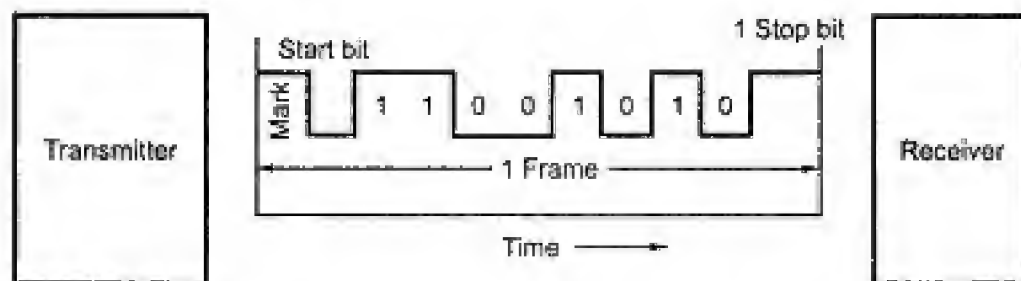


Fig. 14.2 Asynchronous format with data byte CAH

The data rate can be expressed as bits/sec. or characters/sec. The term bits/sec is also called the baud rate. The asynchronous format is generally used in low-speed transmission (less than 20 kbits/sec).

14.3.2 Synchronous

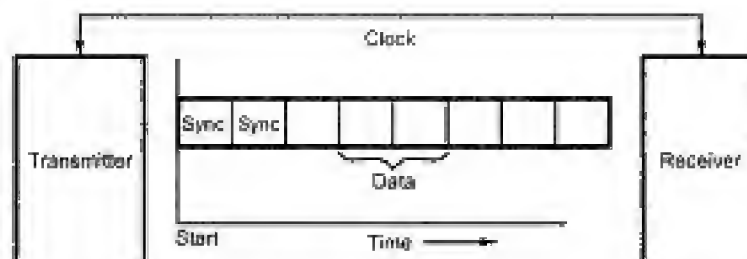


Fig. 14.3 Synchronous transmission format

The start and stop bits in each frame of asynchronous format represents wasted overhead bytes that reduce the overall character rate. These start and stop bits can be eliminated by synchronizing receiver and transmitter. They can be synchronized by having a common clock signal. Such a communication is called **synchronous serial communication**. The Fig. 14.3 shows the transmission format of synchronous serial communication. In this transmission synchronous bits are inserted instead of start and stop bits.

| Sr. No. | Asynchronous serial communication | Synchronous serial communication |
|---------|--|---|
| 1. | Transmitters and receivers are not synchronized by clock. | Transmitter and receivers are synchronized by clock. |
| 2. | Bits of data are transmitted at constant rate. | Data bits are transmitted with synchronisation of clock. |
| 3. | Character may arrive at any rate at receiver. | Character is received at constant rate. |
| 4. | Data transfer is character oriented. | Data transfer takes place in blocks. |
| 5. | Start and stop bits are required to establish communication of each character. | Start and stop bits are not required to establish communication of each character; however, synchronisation bits are required to transfer the data block. |
| 6. | Used in low-speed transmissions at about speed less than 20 kbits/sec. | Used in high-speed transmissions |

Table 14.1 Comparison between asynchronous and synchronous serial data transfer

14.4 RS-232C

In response to the need for signals and handshake standards between DTE and DCE, the Electronic Industries Association (EIA) introduced EIA standard RS-232 in 1962. It was revised and named as RS-232C, in 1969 by EIA. It is widely accepted for single ended data transmission over short distances with low data rates.

This standard describes the functions of 25 signal and handshake pins for serial data transfer. It also describes the voltage levels, impedance levels, rise and fall times, maximum bit rate, and maximum capacitance for these signal lines. RS-232C specifies 25 signal pins and it specifies that the DTE connector should be male, and the DCE connector should be a female. The most commonly used connector should be a female. The most commonly used connector, DB-25P is shown in the Fig. 14.4.

14.4.1 DB-25 P Connector

The Table 14.2 shows pins and signals description for RS 232 C DB-25P connector.

| Pin number | Common name | RS-232C name | Description | Signal direction on DEC |
|------------|-------------|--------------|-------------------|-------------------------|
| 1 | | AA | Protective ground | |
| 2 | TXD | BA | Transmitted data | IN |
| 3 | RXD | BB | Received data | OUT |

| | | | | |
|----|-------------------------|-------|---|--------|
| 4 | $\overline{\text{RTS}}$ | CA | Request to send | IN |
| 5 | $\overline{\text{CTS}}$ | CB | Clear to send | OUT |
| 6 | $\overline{\text{DSR}}$ | CC | Data set ready | OUT |
| 7 | GND | AB | Signal ground (common return) | |
| 8 | $\overline{\text{CD}}$ | CF | Received line signal detector | OUT |
| 9 | | | (Reserved for data set testing) | |
| 10 | | | (Reserved for data set testing) | |
| 11 | | | Unsigned | |
| 12 | | SCF | Secondary recd. line sig. detector | OUT |
| 13 | | SCB | Secondary clear to send | OUT |
| 14 | | SBA | Secondary transmitted data | IN |
| 15 | | DB | Transmission signal element timing (DCE source) | OUT |
| 16 | | SBB | Secondary received data | OUT |
| 17 | | DD | Receiver signal element timing (DCE source) | OUT |
| 18 | | | Unassigned | |
| 19 | | SCA | Secondary request to send | IN |
| 20 | $\overline{\text{DTR}}$ | CD | Data terminal ready | IN |
| 21 | | CG | Signal quality detector | OUT |
| 22 | | CE | Ring indicator | OUT |
| 23 | | CH/CI | Data signal rate selector (DTE/DCE source) | IN/OUT |
| 24 | | DA | Transmit signal element timing (DTE source) | IN |
| 25 | | | Unassigned | |

Table 14.2

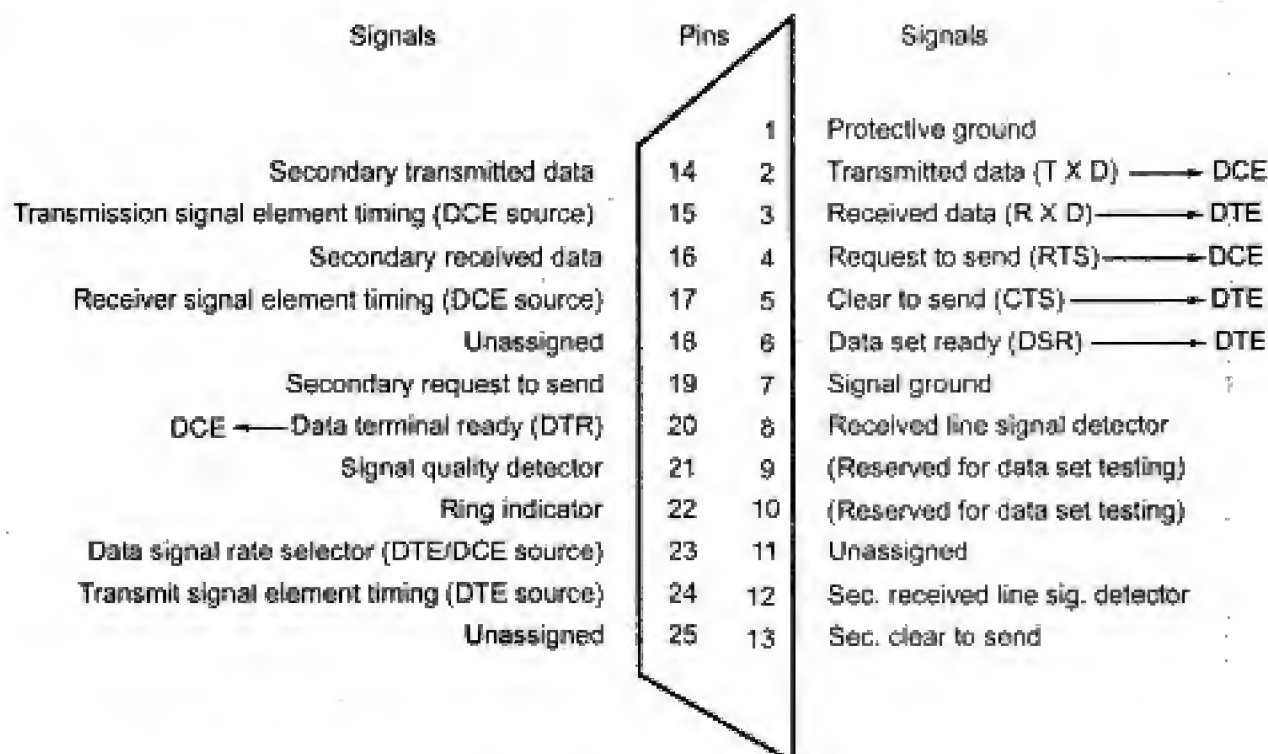


Fig. 14.4

14.4.2 DB-9P Connector

Since not all the pins are used in PC cables, IBM introduced the DB-9P version of the serial I/O standard, which uses only 9-pins. The Table 14.3 describes the pins for DB-9P and Fig. 14.5 shows the DB-9P connector.

| Pin No. | Pin | Description |
|---------|-----|---------------------|
| 1 | DCD | Data carrier detect |
| 2 | RxD | Received data |
| 3 | TxD | Transmitted data |
| 4 | DTR | Data terminal ready |
| 5 | GND | Signal ground |
| 6 | DSR | Data set ready |
| 7 | RTS | Request to send |
| 8 | CTS | Clear to send |
| 9 | RI | Ring Indicator |

Table 14.3 Pin description for DB-9P connector

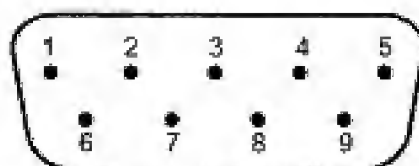


Fig. 14.5 DB-9P connector

14.4.3 Data Transmission and Reception using RS-232C

Fig. 14.6 shows the digital data transmission using modems and standard telephone lines. Modems and other equipment used to send serial data over long distances are known as Data Communication Equipment (DCE). The terminals and computers that are sending or receiving the serial data are referred to as Data Terminal Equipment (DTE).

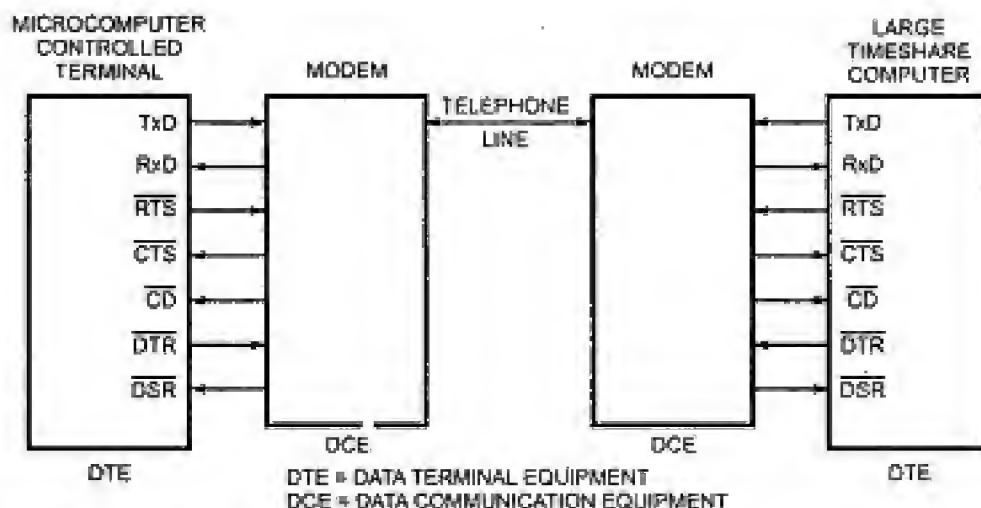


Fig. 14.6 Digital data transmission using modems and standard telephone lines

After the terminal power is turned on and the terminal runs any self-checks, it asserts the data terminal ready ($\overline{\text{DTR}}$) signal to tell the modem it is ready. When modem is ready to transmit or receive data, it asserts the data set ready ($\overline{\text{DSR}}$) signal to the terminal. After receiving $\overline{\text{DSR}}$, the DTE requests use of the data channel by asserting $\overline{\text{RTS}}$ signal to begin transmission. The MODEM at the receiving end is then dialed. This answer-mode modem responds with a 2225 Hz carrier frequency. When the modem connected to the sending terminal receives this carrier, it asserts carrier detect ($\overline{\text{CD}}$) to the terminal. After proper time interval the modem sends clear-to-send signal ($\overline{\text{CTS}}$) indicating modem and communication channels are ready for transmission. Immediately after $\overline{\text{CTS}}$ signal the terminal at the transmitting end sends serial data on its TXD output. A similar handshake takes place for receiving data. Fig. 14.7 shows flowchart for the handshaking process when an RS-232C data port is used to exchange data with the modem.

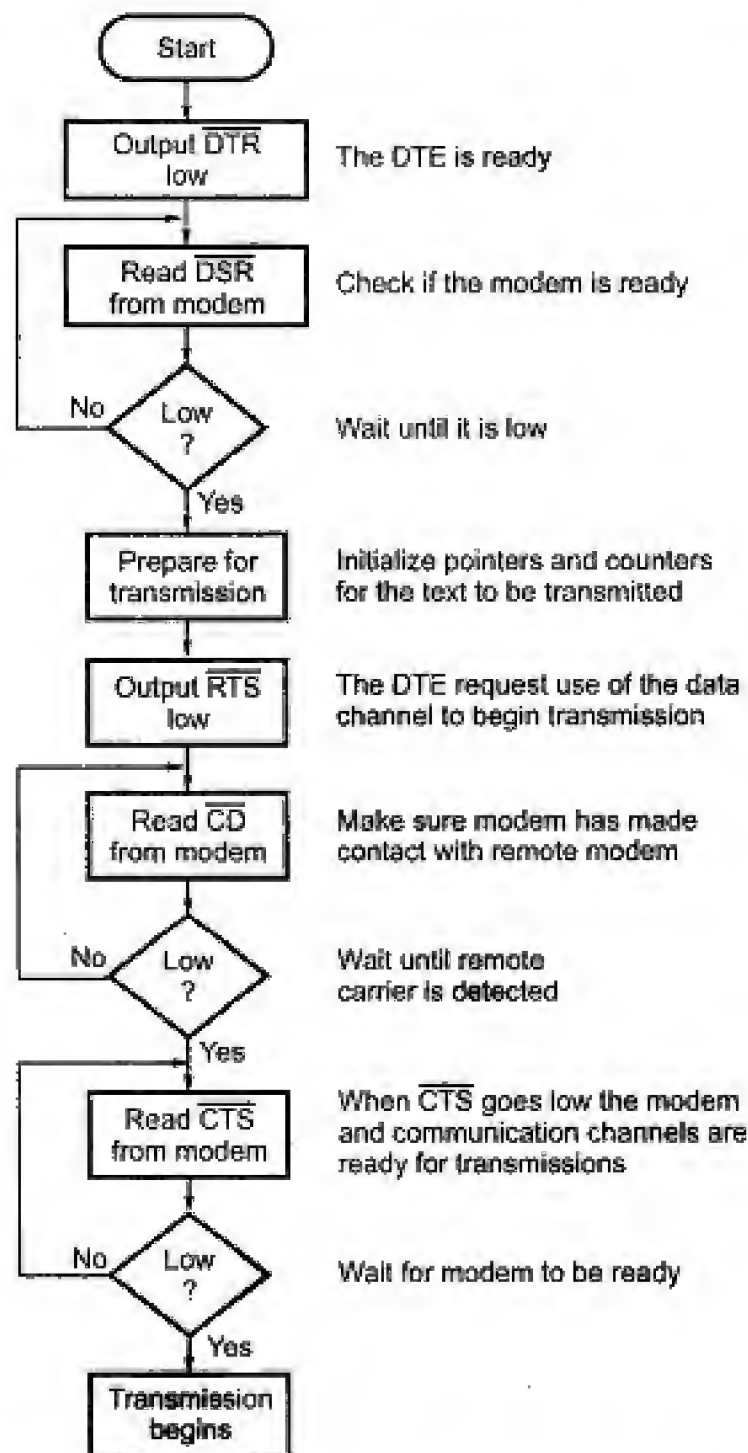


Fig. 14.7 Flowchart showing handshaking process

14.4.4 8051 Connection to RS-232C

In RS-232C the voltage level +3 V to +15 V is defined as logic 0; from -3 V to -15 V is defined as logic 1. The control and timing signals are compatible with the TTL level. Because of the incompatibility of the data lines with the TTL logic, voltage translators,

called line drivers and line receivers, are required to interface TTL logic with the RS-232C signals. The Fig. 14.8 shows the connection between RS-232C and 8051. Here, MAX 232 chip is used as a line driver and line receiver. We know that 8051 assigns two pins RxD (P3.0) and TxD (P3.1) for reception and transmission of serial data, respectively. These pins are TTL compatible; therefore, they require line driver and line receiver to make them RS-232C compatible. The MAX32 has two sets of line drivers and line receivers for transmitting and receiving data. Only one set is required for one serial communication.

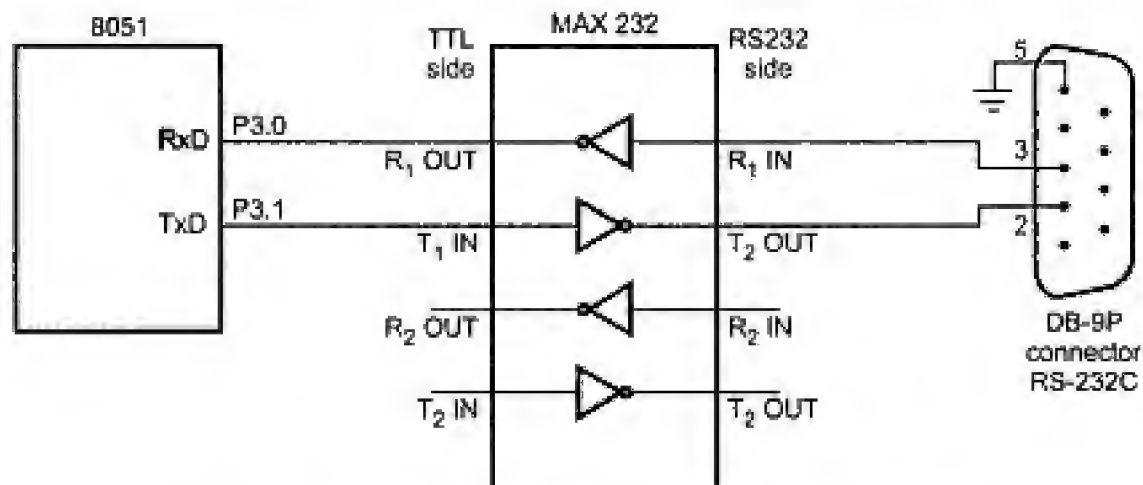


Fig. 14.8 Connection between RS-232C and 8051

14.5 RS-423-A

It is an enhancement of the RS-232C. This standard specifies the electrical characteristics of unbalanced voltage-mode digital interface circuits normally used for the interchange of serial binary signals between DTE and DCE, as shown in the Fig. 14.9.

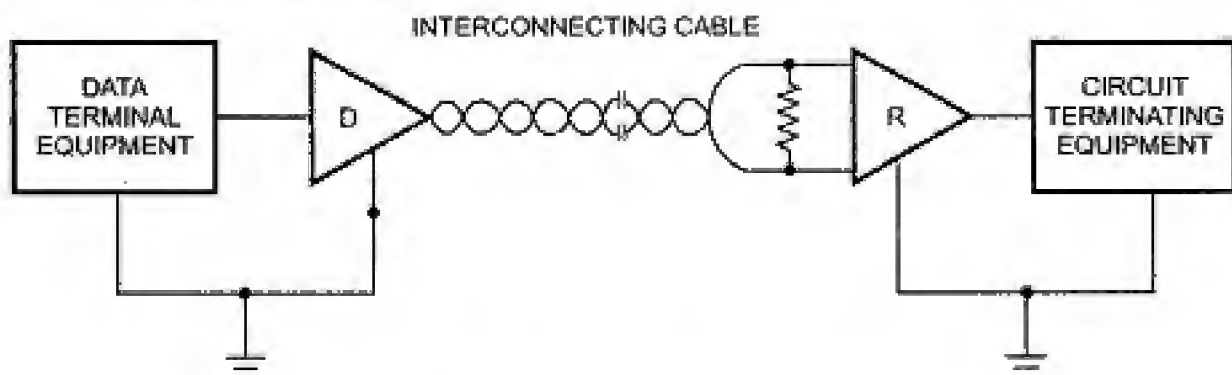


Fig. 14.9 Basic RS-423-A unbalanced digital interface

It allows one driver and up to 10 receivers on a single data line. This standard achieves longer transmission distances (4000 ft) and higher band rates (upto 100 kbps) than the RS-232C standard. Basic requirements for drivers and receivers of RS-423-A standard are listed in Table 14.4 and 14.5.

| Parameter | Specification | |
|---|---------------|---|
| Output drive voltage open circuited output | V_O | $\pm 4 \text{ V to } \pm 6 \text{ V}$ |
| Output drive voltage terminated in 450Ω | V_I | $\geq 0.9 V_O$ |
| Output short circuit current | I_S | $< 150 \text{ mA}$ with output in either logic state |
| Output leakage current (with power off) | I_X | $< 100 \mu\text{A}$ with output voltage of $4\text{V to } 6 \text{ V}$. |
| Output slew rate | S_R | Shall not exceed $15 \text{ V}/\mu\text{s}$ at any point during the transitional period. |
| Output monotonicity | | The output shall be monotonic between 0.1 and $0.9 V_{SS}$. |
| Transitional time | | For pulse widths of 1 ms or greater, the transitional time measured between 0.1 and $0.9 V_{SS}$ shall be between $100 \mu\text{s}$ and $300 \mu\text{s}$. |
| Ringing (overshoot or undershoot) | | After completing a transition from one logic state to the other, the signal voltage should not vary more than 10% of V_{SS} from the steady state value until the next transition occurs. |

Table 14.4 RS-423-A driver requirements

| Parameter | Specification | |
|---|-----------------------|---|
| Input resistance | R_{IN} | $4 \text{ k}\Omega$ minimum under power on or power off conditions. |
| Input sensitivity over an input common-mode range of -7 to 7 V . Referred to as differential input threshold voltage. | V_{TH} | The differential input required to ensure the receiver will correctly assume the intended binary output state is 200 mV . The receiver must also maintain correct operation for differential input voltages from 200 mV to 6 V . |
| Differential input voltage maximum | $V_{ID} (\text{max})$ | The maximum V_{ID} without resulting in damage to the receiver is 12 V . |
| Input balance with any common mode voltage from -7 to 7 V . | | Input voltage balance characteristics shall be such that the receiver will remain in its intended state with a differential voltage of 400 mV applied through $500 \Omega (\pm 1 \%)$ to each input. V_{ID} polarity is reversed for the opposite binary state. |

| | |
|--------------------------------------|---|
| Multiple receivers and total loading | Up to 10 receivers may be connected on a single line. Total loading of multiple receivers and added fail safe circuits must have a resistance of 400 Ω or greater. |
| Recommended grounding | The single ground wire, or common, should be grounded only at the driver end of the data line. |

Table 14.5 RS-423-A receiver requirements

Drivers and Receivers

The SN75156 and μ A9636 are good examples of EIA RS-423-A drivers. The SN75157 and μ A9637A are examples of EIA RS-423-A receivers.

14.6 RS-422-A

High speed data transmission between computer system components and peripherals over a long distance, under high noise conditions is very difficult with single ended drivers and receivers. As an improvement over single ended drivers and receivers EIA standard introduced RS-422-A which uses low impedance differential signals. This standard achieves transmission rates from 100 kbps to 10 Mbps.

A basic RS-422-A circuit has a data line driver, differential transmission line and loads, where a load may consists of one or more receivers (R) and the line termination resistor (R_T), as shown in Fig. 14.10.

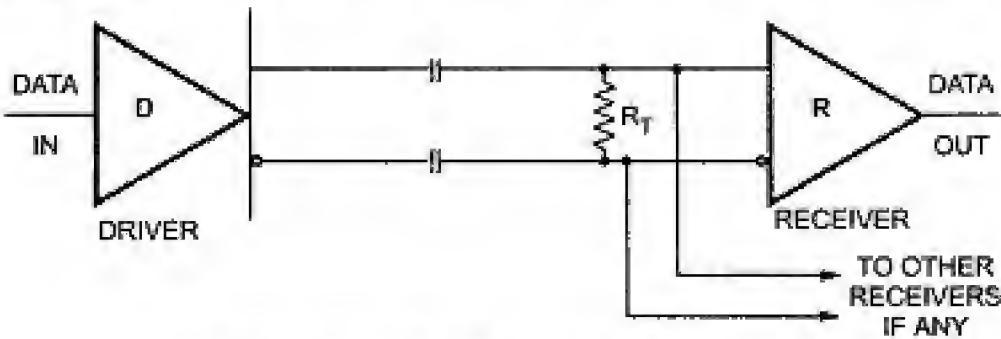


Fig. 14.10 RS-422-A interface circuit

Table 14.6 and 14.7 gives the RS-422-A driver and receiver requirements.

| Parameter | Specification |
|---|----------------------|
| Driver output impedance | 100 Ω or less |
| Output differential voltage | 2.0 V to 6.0 V |
| Difference between opposite polarity differential output voltages | less than 0.4 V |

| | |
|---|---|
| The driver output offset voltage (V_{os}), measured from the junction of the two 50 Ω terminator and driver ground | less than 3.0 V |
| Driver output current | less than 150 mA |
| Output off-state leakage current | less than 100 μ A |
| Output voltage transition time (t_r) | less than 20 ns |
| Overshoot and undershoot magnitudes | less than 10 % of V_{ss} where V_{ss} is the difference between the two steady-state values of the output |

Table 14.6 RS-422-A driver requirements

| Parameter | Specification |
|---|---------------------------------------|
| Differential data input threshold sensitivity | \pm 200 mV |
| Input impedance | Greater than or equal to 4 k Ω |

Table 14.7 RS-422-A receiver requirement

Drivers and Receivers

The SN75172B and SN75175 are driver and receiver for EIA RS-422-A standard, respectively.

14.7 RS-485

EIA standard RS-485, introduced in 1983, is an upgraded version of EIA RS-422-A. Increasing use of balanced data transmission lines in distributing data to several system components and peripherals over relatively long lines brought about the need for multiple driver/receiver combinations on a single twisted pair line.

EIA RS-485 takes into account RS-422-A requirements for balanced-line transmission plus additional features allowing for multiple drivers and receivers. Fig. 14.11 illustrates an application with multiple drivers and receivers.

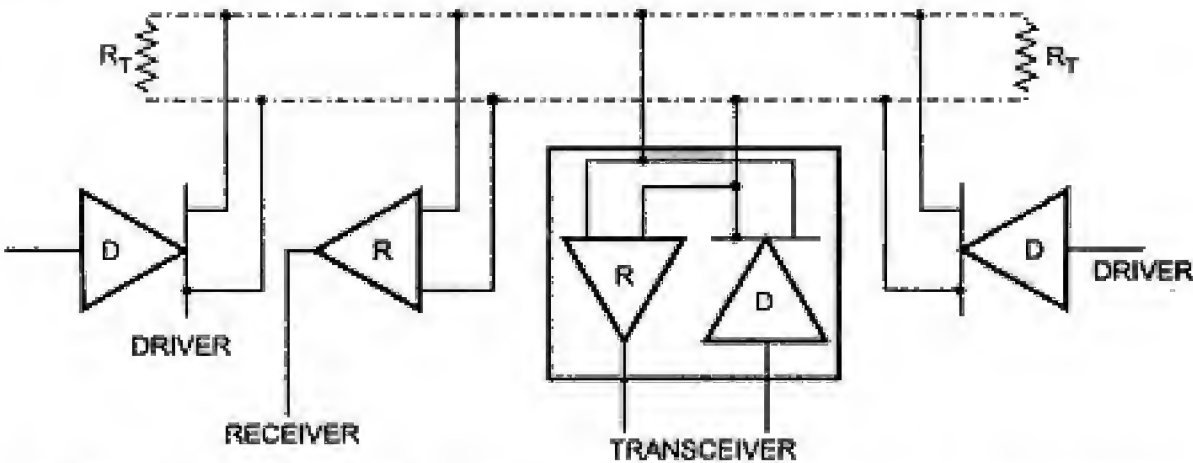


Fig. 14.11 Multiple drivers and receivers interface using EIA RS-485 standard

Standard RS-485 differs from the RS-422-A standard primarily in the features that allow reliable multipoint communications. For the drivers these features are :

- One driver can drive as many as 32 unit loads and a total line termination resistance of 60 Ω or more (one unit load is typically one passive driver and one receiver).
- The driver output, off-state, leakage current shall be 100 μ A or less with any line voltage from - 7 V to 7 V.
- The driver shall be capable of providing a differential output voltage of 1.5 V to 5V with common-mode line voltages from - 7 V to 12 V.
- Drivers must have self protection against contention) multiple drivers contending for the transmission line at the same time). That is, no driver damage shall occur when its outputs are connected to a voltage source of - 7 V to 12 V whether its output state is a binary 1, binary 0 or passive.

For receiver these features are :

- High receiver-input resistance, 12 k Ω minimum.
- A receiver input common-mode range of - 7 V to 12 V.
- Differential input sensitivity of \pm 200 mV over a common-mode range of - 7 V to 12V.

The SN75172B and SN75174B drivers exhibit a differential output transition time of 75 ns maximum. This particular parameter is key to the maximum speed at which the driver may operate in accordance with EIA RS-485 specifications.

14.8 Comparison between EIA Standards

The Table 14.8 shows the comparison between EIA standards

| Parameter | RS-232C | RS-423-A | RS-422-A | RS-485 |
|---|---------------------------------|----------------------------------|--------------------------|----------------------------|
| Mode of operation | Single-ended | Single-ended | Differential | Differential |
| Number of drivers and receivers allowed | 1 Driver 1 Receiver | 1 Driver 10 Receivers | 1 Driver 10 Receivers | 32 Drivers 32 Receivers |
| Maximum cable length (ft) | 50 | 4000 | 4000 | 4000 |
| Maximum data rate bits per second | 20 k | 100 k | 10 M | 10 M |
| Maximum common mode voltage | \pm 25 V | \pm 6 V | 6 V - 0.25 V | 12 V - 7 V |
| Driver output | \pm 5 V min \pm 15 V max | \pm 3.6 V min \pm 6.0 max | \pm 2 V min | \pm 1.5 V min |
| Driver load | 3 k Ω to 7 k Ω | 450 Ω min | 100 Ω min | 60 Ω min |
| Driver slew rate | 30 V μ s max | Externally controlled | NA | NA |

| | | | | |
|---|------------------------------|-------------------|---------------|--|
| Driver output short circuit current limit | 500 mA to V_{CC} or GRD | 150 mA to GRD | 150 mA to GRD | 150 mA to GRD 250 mA to - 8 V or 12 V |
| Driver output resistance Power on | NA | NA | NA | 120 k Ω |
| (High Z state) Power off | 300 Ω | 60 k $\Omega \pm$ | 60 k Ω | 120 k Ω |
| Receiver input resistance Ω | 3 k Ω to 7 k Ω | 4 k Ω | 4 k Ω | 12 k Ω |
| Receiver sensitivity | ± 3 V | ± 200 mV | ± 200 mV | ± 200 mV |

Table 14.8 Comparison between EIA standards

14.9 I²C

Communication network systems are rapidly growing in size and complexity. These systems have many high speed integrated circuits with critical operating parameters and must provide extremely reliable service with zero down time. To maintain the performance of these systems, adequate environmental monitoring must be performed, so a failure or a data trend leading to a potential failure can be rapidly identified. Furthermore, this monitoring must be performed cheaply to keep system costs low.

To provide such needs by maximizing hardware efficiency alongwith providing circuit simplicity, Philips developed a simple bi-directional 2-wire bus for efficient Inter-IC control. This bus is called the Inter IC (Integrated Circuit) bus or I²C bus. The I²C interface employs a comprehensive protocol to ensure reliable transmission and reception of data within connected devices. Here, each device is recognized by a unique address (whether it's a microcontroller, LCD driver, memory or keyboard interface) and can operate as either a transmitter or receiver, depending on the function of the device. Obviously an LCD driver is only a receiver, whereas a memory can both receive and transmit data. In addition to transmitters and receivers, devices can also be considered as masters or slaves when performing data transfers. A master is the device which initiates a data transfer on the bus and generates the clock signals to permit that transfer. At that time, any device addressed is considered as a slave. Let us see some important features of I²C bus.

14.9.1 Features of I²C Bus

The important features of the I²C bus are:

- Only two bus lines are required; a serial data line (SDA) and a serial clock (SCL).
- Each device connected to the bus is software addressable by a unique address and simple master/slave relationships exist at all times; masters can operate as master-transmitters or as master-receivers.

- It's a true multi-master bus including collision detection and arbitration to prevent data corruption if two or more masters simultaneously initiate data transfer.
- Serial, 8-bit oriented, bi-directional data transfers can be made at
- up to 100 kbit/s in the Standard-mode,
- up to 400 kbit/s in the Fast-mode, or
- up to 3.4 Mbit/s in the High-speed mode
- On-chip filtering rejects spikes on the bus data line to preserve data integrity.
- The number of ICs that can be connected to the same bus is limited only by a maximum bus capacitance of 400 pF.

14.9.2 I²C Bus Terminology

The Table 14.9 lists the terminologies used in the I²C bus.

| Term | Description |
|-----------------|---|
| Transmitter | The device that sends the data to the bus. |
| Receiver | The device that receives the data from the bus. |
| Master | The device which initiates the transfer, generates the clock and terminates the transfer. |
| Slave | The device addressed by a master. |
| Multi-master | More than one master device in a system. These masters can attempt to control the bus at the same time without corrupting the message. |
| Arbitration | Procedure that ensures that only one of the master devices will control the bus. This ensure that the transfer data does not get corrupted. |
| Synchronization | Procedure where the clock signals of two or more devices are synchronized. |

Table 14.9

14.9.3 I²C BUS Configuration

Let us see the example of I²C bus configuration using two microcontrollers. The I²C bus is a multi-master bus. This means that more than one device capable of controlling the bus can be connected to it. The Fig 14.12 shows the two microcontrollers connected to I²C bus. Due to two microcontrollers, there exists master-slave and receiver-transmitter relationships on the I²C bus. It should be noted that these relationships are not permanent, but only depend on the direction of data transfer at that time. The transfer of data would proceed as follows :

Case 1 : Microcontroller A wants to send information to microcontroller B

- microcontroller A (master), addresses microcontroller B (slave).

- microcontroller A (master-transmitter), sends data to microcontroller B (slave-receiver).
- microcontroller A terminates the transfer.

Case 2 : Microcontroller A wants to receive information from microcontroller B

- microcontroller A (master) addresses microcontroller B (slave).
- microcontroller A (master- receiver) receives data from microcontroller B (slave-transmitter).
- microcontroller A terminates the transfer.

Even in this case, the master (microcontroller A) generates the timing and terminates the transfer.

The possibility of connecting more than one microcontroller to the I²C bus means that more than one master could try to initiate a data transfer at the same time. To avoid the chaos an arbitration procedure has been developed.

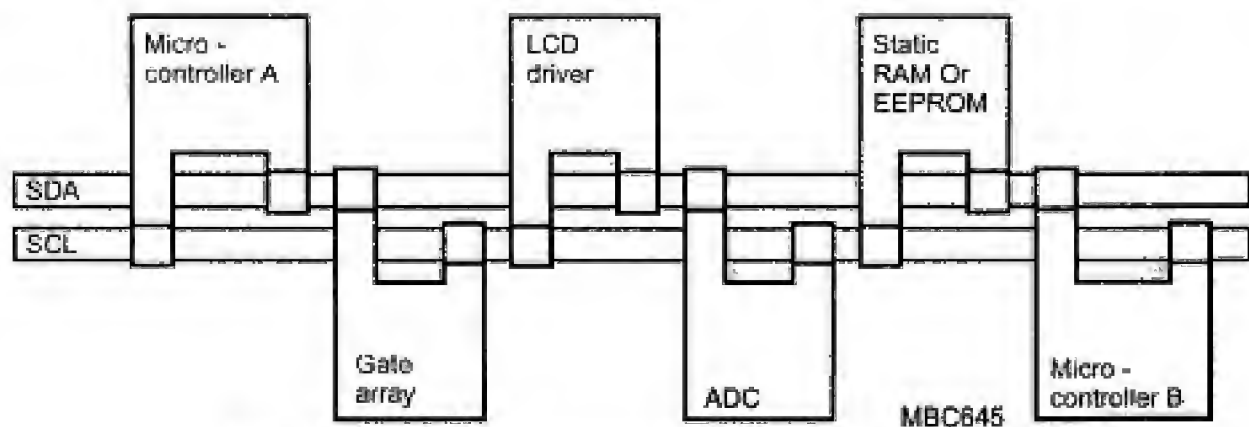


Fig. 14.12 Two microcontrollers connected to I²C bus

14.9.4 I²C Signals

- Start** – high-to-low transition of the SDA line while SCL line is high.
- Stop** – low-to-high transition of the SDA line while SCL line is high.
- Ack** – receiver pulls SDA low while transmitter allows it to float high.
- Data** – transition takes place while SCL is low.

14.9.5 Initiating and Terminating Data Transfer

During times of no data transfer (idle time), both the clock line (SCL) and the data line (SDA) are pulled high through the external pull-up resistors. The START and STOP conditions determine the start and stop of data transmission. The START condition is defined as a high to low transition of the SDA when the SCL is high. The STOP condition is defined as a low to high transition of the SDA when the SCL is high. Fig. 14.13 shows

the START and STOP conditions. The master generates these conditions for starting and terminating data transfer. Due to the definition of the START and STOP conditions, when data is being transmitted, the SDA line can only change state when the SCL line is low.

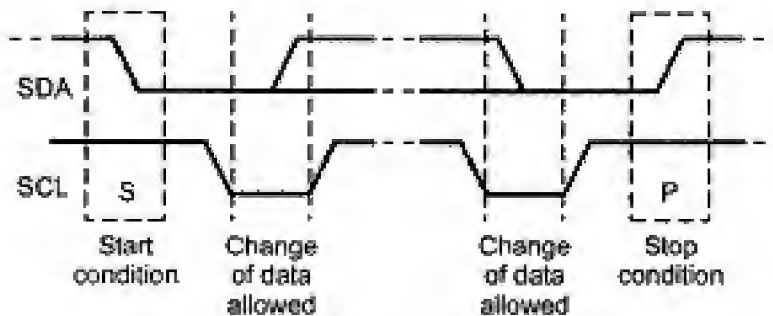


Fig. 14.13 Start and stop conditions

START and STOP conditions are always generated by the master. The bus is considered to be busy after the START condition. The bus is considered to be free again a certain time after the STOP condition.

The bus stays busy if a repeated START (Sr) is generated instead of a STOP condition. In this respect, the START (S) and repeated START (Sr) conditions are functionally identical. Thus, hereafter, the S symbol will be used as a generic term to represent both the START and repeated START conditions, unless Sr is particularly relevant.

Detection of START and STOP conditions by devices connected to the bus is easy if they incorporate the necessary interfacing hardware. However, microcontrollers with no such interface have to sample the SDA line at least twice per clock period to sense the transition.

14.9.6 Transferring Data

14.9.6.1 Byte Format

In the byte format, 8-bit data is put on the SDA line. The number of bytes that can be transmitted per transfer is unrestricted. Each byte has to be followed by an acknowledge bit. Data is transferred with the most significant bit (MSB) first as shown in Fig. 14.14. If a slave can't receive or transmit another complete byte of data until it has performed some

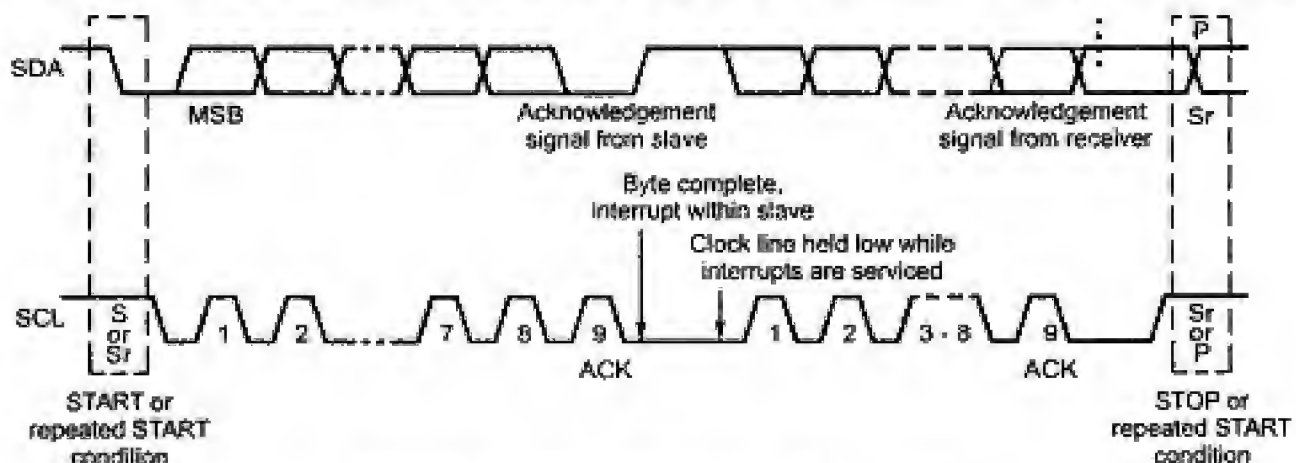


Fig. 14.14 Data transfer on the I²C bus

other function, for example servicing an internal interrupt, it can hold the clock line SCL LOW to force the master into a wait state. Data transfer then continues when the slave is ready for another byte of data and releases clock line SCL.

A message which starts with such an address can be terminated by generation of a STOP condition, even during the transmission of a byte. In this case, no acknowledge is generated.

14.9.6.2 Acknowledge

Data transfer with acknowledge is compulsory. The acknowledge-related clock pulse is generated by the master. The transmitter releases the SDA line (HIGH) during the acknowledge clock pulse. The receiver must pull down the SDA line during the acknowledge clock pulse so that it remains stable LOW during the HIGH period of this clock pulse as shown in Fig. 14.15. Of course, set-up and hold times must also be taken into account.

When a slave doesn't acknowledge the slave address (for example, it's unable to receive or transmit because it's performing some real-time function), the data line must be left HIGH by the slave. The master can then generate either a STOP condition to abort the transfer, or a repeated START condition to start a new transfer.

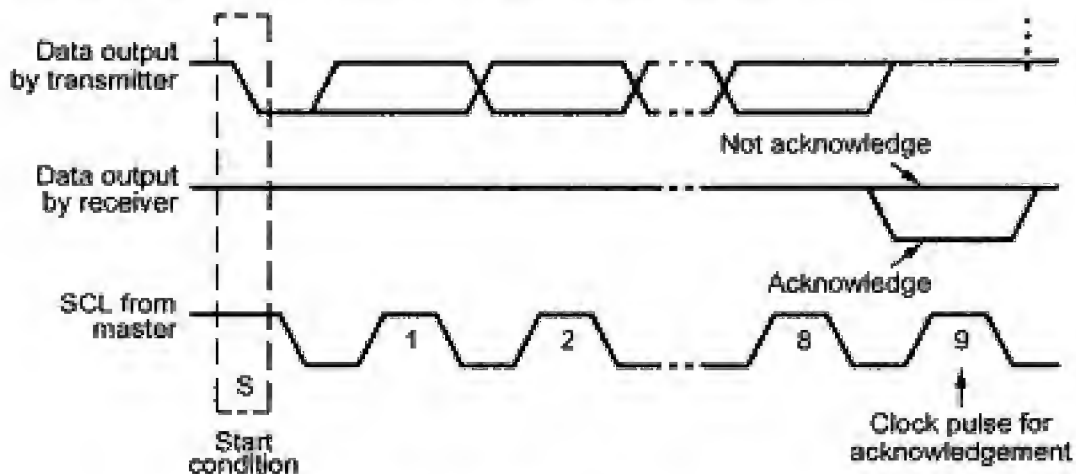


Fig. 14.15 Acknowledge on the I²C bus

If a master-receiver is involved in a transfer, it must signal the end of data to the slave-transmitter by not generating an acknowledge on the last byte that was clocked out of the slave. The slave-transmitter must release the data line to allow the master to generate a STOP or repeated START condition.

14.9.7 Addressing I²C Devices

There are two address formats. The simplest is the 7-bit address format with a R/W bit. The more complex is the 10-bit address with a R/W bit. For 10-bit address format, two bytes must be transmitted with the first five bits specifying this to be a 10-bit address. The 10-bit addressing allows the use of up to 1024 additional slave addresses. The 10-bit

addressing does not affect the existing 7-bit addressing. Devices with 7-bit and 10-bit addresses can be connected to the same I²C bus.

The Fig. 14.16 shows the 7-bit address format and Fig. 14.17 shows the 10-bit address format.

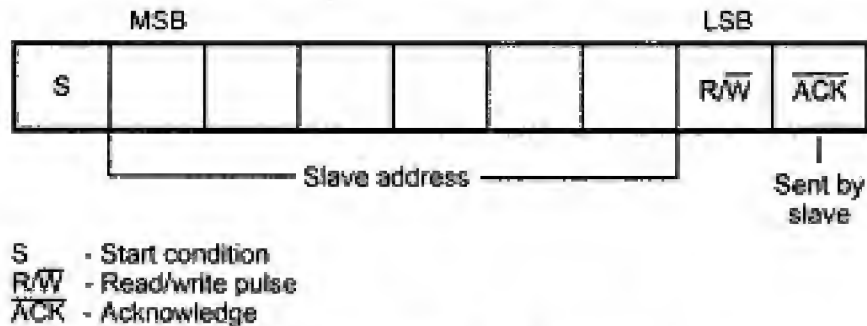


Fig. 14.16 7-bit address format

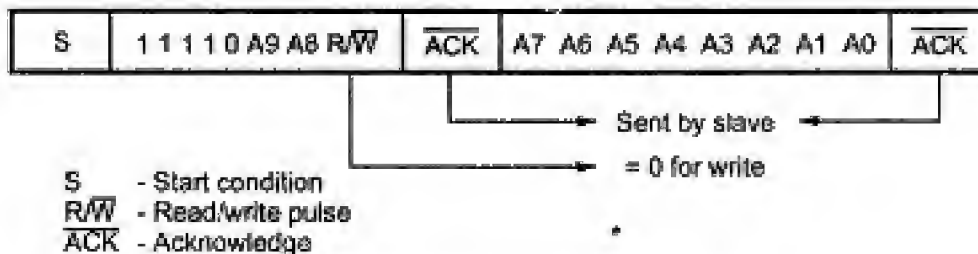


Fig. 14.17 10-bit address format

14.9.8 Complete Data Transfer

Fig. 14.18 shows complete data transfer using 7-bit address format. As shown in Fig. 14.18, after the START condition (S), a slave address is sent. This address is 7 bits long followed by an eighth bit which is a data direction bit (R/W) - a 'zero' indicates a transmission (WRITE), a 'one' indicates a request for data (READ). A data transfer is always terminated by a STOP condition (P) generated by the master. However, if a master still wishes to communicate on the bus, it can generate a repeated START condition (Sr) and address another slave without first generating a STOP condition. When an address is sent, each device in a system compares the first seven bits after the START condition with its address. If they match, the device considers itself addressed by the master as a slave-receiver or slave-transmitter, depending on the R/W bit. Various combinations of read/write formats are then possible within such a transfer.

The possible data transfer formats are as follows :

1. Master-transmitter transmits to slave-receiver. The transfer direction is not changed. This is shown in Fig. 14.18

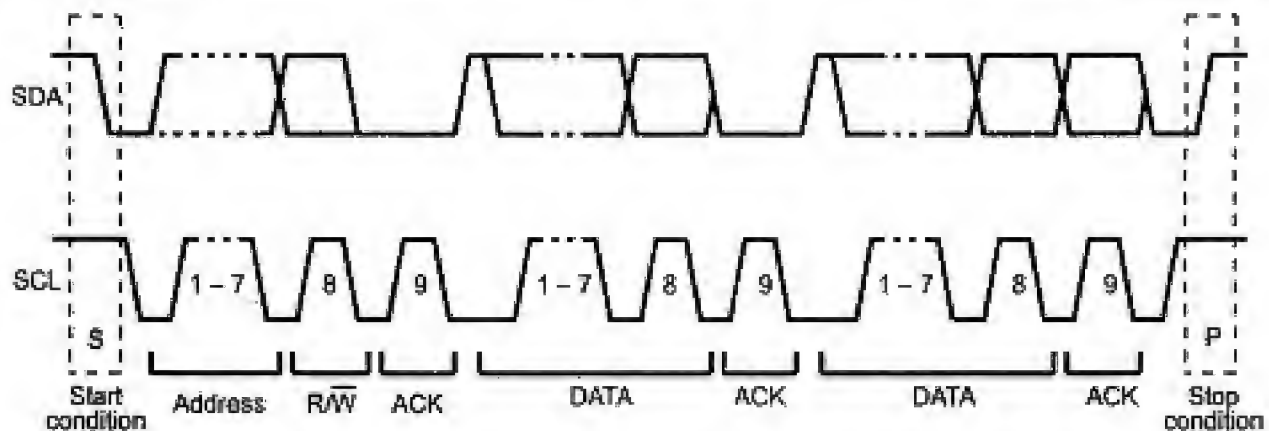


Fig. 14.18 A complete data transfer using 7-bit address format

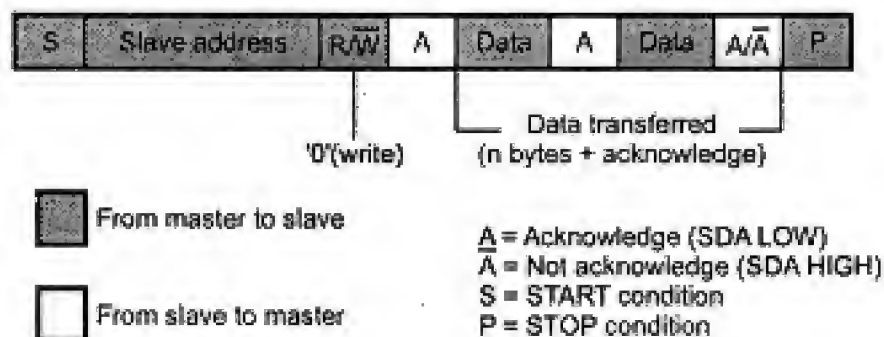


Fig. 14.19 Master-transmitter transmits to slave-receiver

2. Master reads slave immediately after first byte. This is shown in Fig 14.20. At the moment of the first acknowledge, the master-transmitter becomes a master-receiver and the slave-receiver becomes a slave-transmitter. This first acknowledge is still generated by the slave. The STOP condition is generated by the master, which has previously sent a not-acknowledge (A).

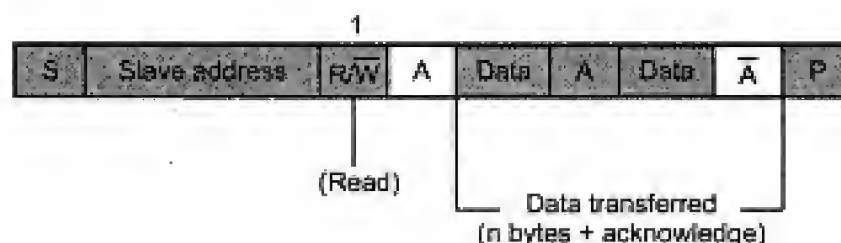
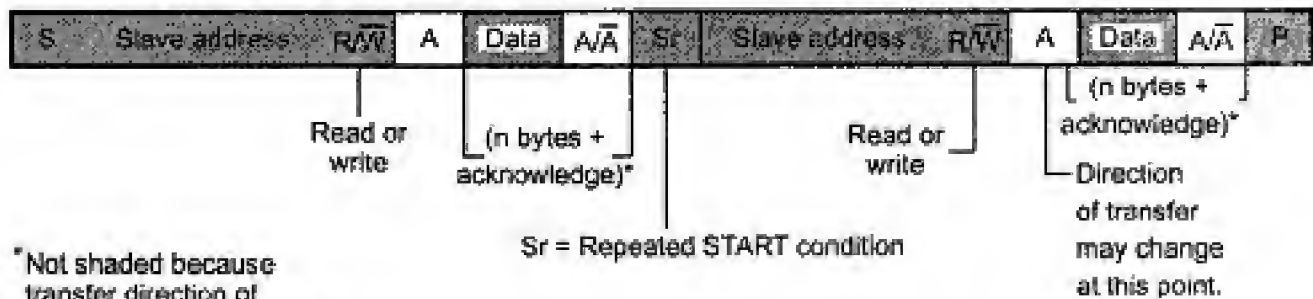


Fig. 14.20 Master reads slave immediately after first byte

3. The Fig. 14.21 shows the combined format. During a change of direction within a transfer, the START condition and the slave address are both repeated, but with the R/W bit reversed. If a master receiver sends a repeated START condition, it has previously sent a not-acknowledge (A).



*Not shaded because transfer direction of data and acknowledge bits depends on R/W bits.

Fig. 14.21 Combined format.

14.9.9 Arbitration

A master may start a transfer only if the bus is free. Two or more masters may generate a START condition within the minimum hold time ($t_{HD:STA}$) of the START condition which results in a defined START condition to the bus. Arbitration takes place on the SDA line, while the SCL line is at the HIGH level, in such a way that the master which transmits a HIGH level, while another master is transmitting a LOW level will switch off its DATA output stage because the level on the bus doesn't correspond to its own level. This is illustrated in Fig. 14.22. As shown in the Fig. 14.22, at point A, master 1 transmits HIGH level and master 2 transmits LOW level. Since SDA line uses wired-AND logic the effective level on the SDA line is LOW. Thus, master 1 level does not match with the SDA level and it switches off its DATA output stage.

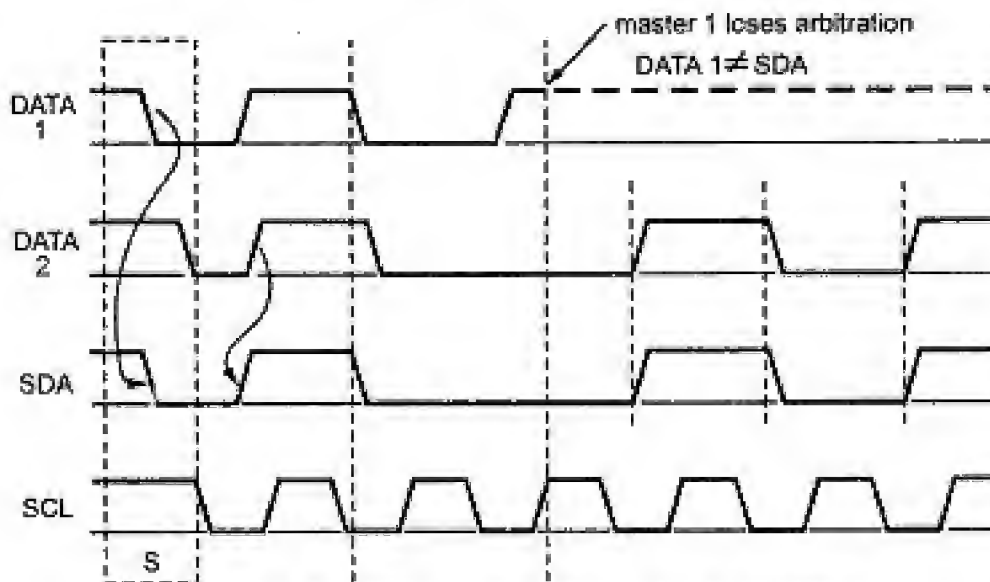


Fig. 14.22 Arbitration procedure of two masters.

If more than one masters are trying to address the same device, arbitration continues with comparison of the data-bits if they are master-transmitter, or acknowledge-bits if they are master-receiver. Because address and data information on the I²C bus is determined by the winning master, no information is lost during the arbitration process.

It is important to note that, arbitration isn't allowed between :

- A repeated START condition and a data bit
- A STOP condition and a data bit
- A repeated START condition and a STOP condition.

Note : Slaves are not involved in the arbitration procedure.

14.9.10 Extensions to the Standard-Mode I²C Bus Specification

The Standard-mode I²C bus specification, with its data transfer rate of up to 100 kbit/s and 7-bit addressing, has been in existence since the beginning of the 1980's. To meet the demands for higher speeds, as well as make available more slave addresses for the growing number of new devices, the Standard-mode I²C bus specification was upgraded over the years and today is available with the following extensions :

- **Fast-mode**, with a bit rate up to 400 kbit/s.
- **High-speed mode (Hs-mode)**, with a bit rate up to 3.4 Mbit/s.
- **10-bit addressing**, which allows the use of up to 1024 additional slave addresses.

Let us see the details of fast mode and High-speed mode. We have already discussed 10-bit addressing in section 14.9.7.

14.9.10.1 Fast-mode

Fast-mode devices can receive and transmit at upto 400 kbit/s. The minimum requirement is that they can synchronize with a 400 kbit/s transfer; they can then prolong the LOW period of the SCL signal to slow down the transfer. Fast-mode devices are downward-compatible and can communicate with standard-mode devices in a 0 to 100 kbit/s I²C bus system.

Features :

The Fast-mode I²C bus specification has the following additional features compared with the standard-mode:

- The maximum bit rate is increased to 400 kbit/s.
- Timing of the serial data (SDA) and serial clock (SCL) signals has been adapted. There is no need for compatibility with other bus systems such as CBUS because they cannot operate at the increased bit rate.
- The inputs of Fast-mode devices incorporate spike suppression and a Schmitt trigger at the SDA and SCL inputs.

- The output buffers of Fast-mode devices incorporate slope control of the falling edges of the SDA and SCL signals.
- If the power supply to a Fast-mode device is switched off, the SDA and SCL I/O pins must be floating so that they don't obstruct the bus lines.
- The external pull-up devices connected to the bus lines must be adapted to accommodate the shorter maximum permissible rise time for the Fast-mode I²C bus.

14.9.10.2 Hs-mode

Hs-mode devices can transfer information at bit rates of up to 3.4 Mbit/s, yet they remain fully downward compatible with Fast- or standard-mode (F/S-mode) devices for bi-directional communication in a mixed-speed bus system. With the exception that arbitration and clock synchronization is not performed during the Hs-mode transfer, the same serial bus protocol and data format is maintained as with the F/S-mode system.

Improvements made in the regular I²C-bus specification to achieve a bit transfer of up to 3.4 Mbit/s are as follows :

- Hs-mode master devices have an open-drain output buffer for the SDAH signal and a combination of an open-drain pull-down and current-source pull-up circuit on the SCLH output. This current-source circuit shortens the rise time of the SCLH signal.
- No arbitration or clock synchronization is performed during Hs-mode transfer in multi-master systems, which speeds-up bit handling capabilities.
- Hs-mode master devices generate a serial clock signal with a HIGH to LOW ratio of 1 to 2. This relieves the timing requirements for set-up and hold times.
- The reduction in capacitive loads of the SDAH and SCLH lines results in faster rise and fall times.
- The inputs of Hs-mode devices incorporate spike suppression and a Schmitt trigger at the SDAH and SCLH inputs.
- The output buffers of Hs-mode devices incorporate slope control of the falling edges of the SDAH and SCLH signals.

14.9.11 Advantages of I²C

- Good for communication with on-board devices that are accessed occasionally.
- Easy to link multiple devices because of addressing scheme.
- Cost and complexity do not scale up with the number of devices.

14.9.12 Disadvantage of I²C

- The complexity of supporting software components can be higher than that of competing schemes (for example, SPI).

14.9.13 Applications of I²C

- Used as a control interface to signal processing devices that have separate data interfaces, e.g. RF tuners, video decoders and encoders, and audio processors.

14.10 Modbus Protocol

Programmable controllers can communicate with each other and with other devices over a variety of networks. These networks are accessed by built-in ports in the controllers or by network adapters, option modules, and gateways. Modbus protocol defines a message structure that controllers will recognize and use, regardless of the type of networks over which they communicate. It describes –

- the process a controller uses to request access to another device
- how it will respond to requests from the other devices
- a common format for the layout and contents of message fields
- the internal standard that the Modicon controllers use for parsing messages
- how each controller will know its device address, recognize a message addressed to it, determine the kind of action to be taken, and extract any data or other information contained in the message.
- how errors will be detected and reported.

For networks other than Modicon Modbus and Modbus Plus industrial networks such as MAP and Ethernet, messages containing Modbus protocol are imbedded into the frame or packet structure that is used on the network.

14.10.1 Transactions on Modbus Networks

Standard Modbus ports on Modicon controllers use an RS-232C compatible serial interface that defines connector pinouts, cabling, signal levels, transmission baud rates, and parity checking.

Controllers can be networked directly or via modems. Controllers communicate using a master-slave technique, in which only one device (the master) can initiate transactions (called 'queries'). The other devices (the slaves) respond by supplying the requested data to the master, or by taking the action requested in the query. Typical master devices include host processors and programming panels. Typical slaves include programmable controllers.

The master can address individual slaves, or can initiate a broadcast message to all slaves. Slaves return a message (called a 'response') to queries that are addressed to them individually. The slave's response message is also constructed using Modbus protocol. It contains fields confirming the action taken, any data to be returned, and an error-checking field. If an error occurred in receiving the message, or if the slave is unable to perform the requested action, the slave will construct an error message and send it as its response.

14.10.2 The Query-Response Cycle

The Fig. 14.23 shows the query-response cycle.

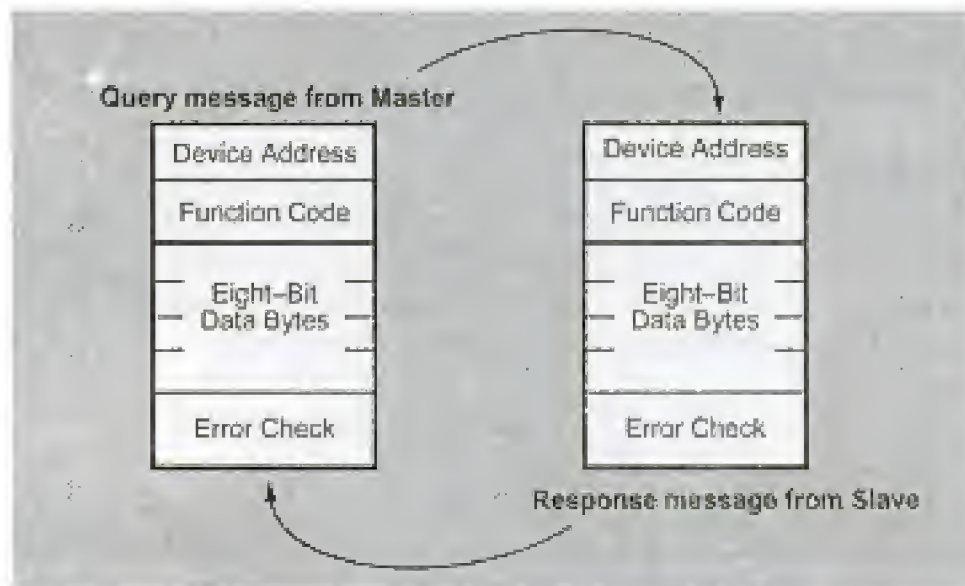


Fig. 14.23 Master-Slave Query-Response Cycle

Query : Device field of the query gives the address of the slave. The function code in the query tells the addressed slave device what kind of action to perform. The data bytes contain any additional information that the slave will need to perform the function. For example, function code 03 will query the slave to read holding registers and respond with their contents. The data field must contain the information telling the slave which register to start at and how many registers to read. The error check field provides a method for the slave to validate the integrity of the message contents.

The Response : If the slave makes a normal response, the function code in the response is an echo of the function code in the query. The data bytes contain the data collected by the slave, such as register values or status. If an error occurs, the function code is modified to indicate that the response is an error response, and the data bytes contain a code that describes the error. The error check field allows the master to confirm that the message contents are valid.

14.10.3 Transmission Modes

Controllers can be setup to communicate on standard Modbus networks using either of two serial transmission modes :

- ASCII or
- RTU

Users can select the desired mode, along with the serial port communication parameters (baud rate, parity mode, etc.), during configuration of each controller. It is important to note that, the mode and serial parameters must be same for all devices on a Modbus network. The selection of ASCII or RTU mode pertains only to standard Modbus networks. These transmission modes define :

- the bit contents of message fields transmitted serially on those networks.
- how information will be packed into the message fields and decoded.

14.10.3.1 ASCII Mode

In ASCII (American Standard Code for Information Interchange) mode, each 8-bit byte in a message is sent as two ASCII characters. The main advantage of this mode is that it allows time intervals of upto one second to occur between characters without causing an error. The format for each byte in ASCII mode is explained in the Fig. 14.24 and it is shown in Table 14.10.

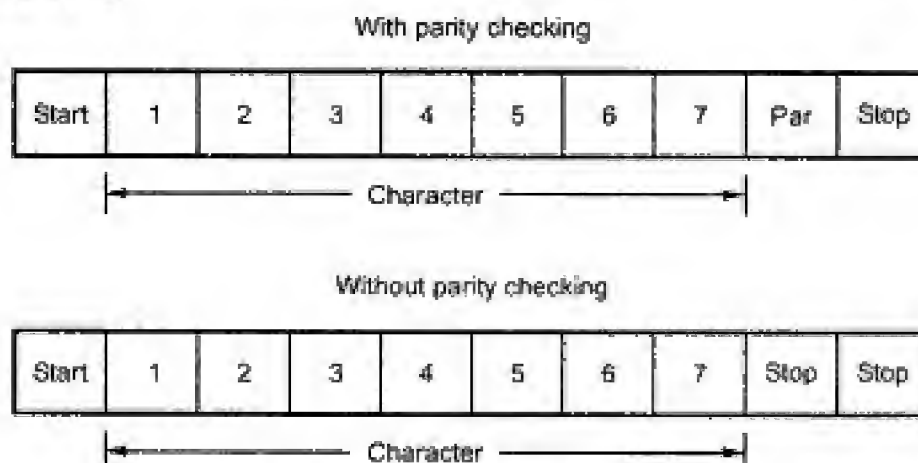


Fig. 14.24 Format for byte in ASCII mode

| Parameter | Format |
|-------------------|--|
| Coding System | Hexadecimal, ASCII characters 0-9, A-F One hexadecimal character contained in each ASCII character of the message |
| Bits per Byte | 1 start bit 7 data bits, least significant bit sent first 1 bit for even/odd parity; no bit for no parity 1 stop bit if parity is used; 2 bits if no parity |
| Error Check Field | Longitudinal Redundancy Check (LRC) |

Table 14.10

14.10.3.2 RTU Mode

In RTU (Remote Terminal Unit) mode, each 8-bit byte in a message contains two 4-bit hexadecimal characters. The main advantage of this mode is that its greater character density allows better data throughput than ASCII for the same baud rate. Each message must be transmitted in a continuous stream. The format for each byte in RTU mode is explained in the Table 14.11 and it is shown in Fig. 14.25.

| Parameter | Format |
|-------------------|--|
| Coding System | 8-bit binary, hexadecimal 0-9, A-F Two 4-bit hexadecimal characters contained in each 8-bit field of the message |
| Bits per Byte | 1 start bit 8 data bits, least significant bit sent first 1 bit for even/odd parity; no bit for no parity 1 stop bit if parity is used; 2 bits if no parity |
| Error Check Field | Cyclical Redundancy Check (CRC) |

Table 14.11

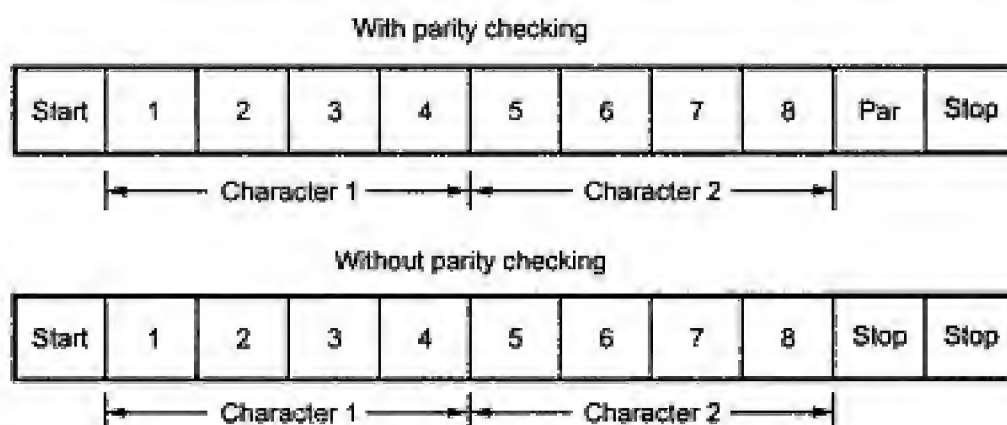


Fig. 14.25 Format for byte in RTU mode

14.10.4 Modbus Message Framing

In either of the two serial transmission modes (ASCII or RTU), a Modbus message is placed by the transmitting device into a frame that has a known beginning and ending point. This allows receiving devices to begin at the start of the message, read the address portion and determine which device is addressed (or all devices, if the message is broadcast), and to know when the message is completed. Partial messages can be detected and errors can be set as a result.

14.10.4.1 ASCII Framing

In ASCII mode, messages start with a 'colon' (:) character (ASCII 3A hex), and end with a 'carriage return - line feed' (CRLF) pair (ASCII 0D and 0A hex respectively).

For all other fields hexadecimal characters 0-9, A-F are allowed to transmit. Networked devices monitor the network bus continuously for the 'colon' character. When it is received, each device decodes the next field (the address field) to find out if it is the addressed device. Intervals of upto one second can elapse between characters within the message. If a greater interval occurs, the receiving device assumes an error has occurred. The Table 14.12 shows a typical message frame.

| START | ADDRESS | FUNCTION | DATA | LRC CHECK | END |
|--------|---------|----------|--------|-----------|----------------|
| 1 CHAR | 2 CHAR | 2 CHAR | n CHAR | 2 CHAR | 2 CHAR CRLF |

Table 14.12 ASCII message frame

14.10.4.2 RTU Framing

In RTU mode, messages start with a silent interval of at least 3.5 character times. This can be implemented as a multiple of character times at the baud rate that is being used on the network (shown as T1-T2-T3-T4 in the Table 14.13).

| START | ADDRESS | FUNCTION | DATA | CRC CHECK | END |
|-------------|---------|----------|------------|-----------|-------------|
| T1-T2-T3-T4 | 8 BITS | 8 BITS | n × 8 BITS | 16 BITS | T1-T2-T3-T4 |

Table 14.13 RTU message frame

The first field then transmitted is the device address. For all other fields hexadecimal characters 0-9, A-F are allowed to transmit. Networked devices monitor the network bus continuously, including during the 'silent' intervals. When the first field (the address field) is received, each device decodes it to find out if it is the addressed device. Following the last transmitted character, a similar interval of at least 3.5 character times marks the end of the message. A new message can begin after this interval.

The entire message frame must be transmitted as a continuous stream. If a silent interval of more than 1.5 character times occurs before completion of the frame, the receiving device flushes the incomplete message and assumes that the next byte will be the address field of a new message.

Similarly, if a new message begins earlier than 3.5 character times following a previous message, the receiving device will consider it a continuation of the previous message. This will set an error, as the value in the final CRC field will not be valid for the combined messages. A typical message frame is as shown in Table 14.13.

14.10.4.3 More About Address Field

We know that, the address field of a message frame contains two characters (ASCII) or eight bits (RTU). Valid slave device addresses for address field which are in the range of 0- 247 decimal. The individual slave devices are assigned addresses in the range of

1 – 247: A master addresses a slave by placing the slave address in the address field of the message. When the slave sends its response, it places its own address in this address field of the response to let the master know which slave is responding. Address 0 is used for the broadcast address, which all slave devices recognize.

14.10.4.4 More About Function Field

We know that, the function code field of a message frame contains two characters (ASCII) or eight bits (RTU). Valid codes for the function field are in the range of 1 – 255 decimal. In which, some codes are applicable to all Modicon controllers, while some codes apply only to certain models, and others are reserved for future use. When a message is sent from a master to a slave device, the function code field tells the slave what kind of action to perform. Examples are :

- to read the ON/OFF states of a group of discrete coils or inputs,
- to read the data contents of a group of registers,
- to read the diagnostic status of the slave,
- to write to designated coils or registers, or
- to allow loading, recording or verifying the program within the slave.

When the slave responds to the master, it uses the function code field to indicate either a normal (error-free) response or that some kind of error occurred (called an exception response). For a normal response, the slave simply sends the original function code. For an exception response, the slave returns the original function code with its most-significant bit set to a logic 1.

For example, if master sends the 0000 0011 (Hexadecimal 03) as a function code to slave then –

- If the slave device takes the requested action without error, it returns the same code in its response.
- If an exception occurs, it returns : 1000 0011 (Hexadecimal 83). In addition to its modification of the function code for an exception response, the slave places a unique code into the data field of the response message. This tells the master what kind of error occurred or the reason for the exception.

14.10.4.5 More About Data Field

The data field is constructed using sets of two hexadecimal digits, in the range of 00 to FF hexadecimal. These can be made from a pair of ASCII characters, or from one RTU character, according to the network's serial transmission mode.

The data field of messages sent from a master to slave devices contains additional information which the slave must use to take the action defined by the function code. This can include items like discrete and register addresses, the quantity of items to be handled, and the count of actual data bytes in the field.

For example, if the master requests a slave to read a group of holding registers (function code 03), the data field specifies the starting register and how many registers are to be read. If the master writes to a group of registers in the slave (function code 10 hexadecimal), the data field specifies the starting register, how many registers to write, the count of data bytes to follow in the data field, and the data to be written into the registers. If no error occurs, the data field of a response from a slave to a master contains the data requested. If an error occurs, the field contains an exception code that the master application can use to determine the next action to be taken.

In certain kinds of messages, the function code alone specifies the action. In such messages, the length of the data field can be zero.

14.10.4.6 More About Error Checking Field

The error checking field contents depend upon the transmission method that is being used either ASCII or RTU.

ASCII

In ASCII mode, the error checking field contains two ASCII characters. The error check characters are the result of a Longitudinal Redundancy Check (LRC) calculation that is performed on the message contents. The beginning 'colon' and terminating CRLF characters are not considered in the Longitudinal Redundancy Check. The LRC characters are appended to the message as the last field preceding the CRLF characters.

RTU

In RTU mode, the error checking field contains a 16-bit value implemented as two 8-bit bytes. The error check value is the result of a Cyclical Redundancy Check calculation performed on the message contents. The CRC field is appended (low-order byte of the field is first followed by the high-order byte) to the message as the last field in the message.

14.10.5 Error Checking Methods

Standard Modbus serial networks use two kinds of error checking :

Parity checking : It (even or odd) can be optionally applied to each character.

Frame checking : It (LRC or CRC) is applied to the entire message.

Both the character check and message frame checks are generated in the master device and applied to the message contents before transmission. The slave device checks each character and the entire message frame during reception.

14.10.5.1 Parity Checking

We can configure controllers for – Even or Odd Parity checking, or for No Parity checking. The parity bit will then be set to a 0 or 1 to result in an Even or Odd total of 1 bits.

When the message is transmitted, the parity bit is determined and applied to the frame of each character. The receiving device counts the quantity of 1 bits and sets an error if they are not the same as configured for that device (all devices on the Modbus network must be configured to use the same parity check method).

If No Parity checking is specified, no parity bit is transmitted and no parity check can be made. An additional stop bit is transmitted to fill out the character frame.

14.10.5.2 Frame Checking

For checking the frame in the ASCII mode we have to perform the LRC check and in the RTU mode we have to perform CRC check.

LRC Checking : The LRC field checks the contents of the message, exclusive of the beginning 'colon' and ending CRLF pair. It is applied regardless of any parity check method used for the individual characters of the message.

The LRC field is one byte, containing an 8-bit binary value. The LRC value is calculated by the transmitting device, which appends the LRC to the message. The receiving device calculates an LRC during reception of the message, and compares the calculated value to the actual value it received in the LRC field. If the two values are not equal, an error results.

The LRC is calculated by adding together successive 8-bit bytes of the message, discarding any carries, and then two's complementing the result. It is performed on the ASCII message field contents excluding the 'colon' character that begins the message, and excluding the CRLF pair at the end of the message.

CRC Checking : The CRC field checks the contents of the entire message. It is applied regardless of any parity check method used for the individual characters of the message.

The CRC field is two bytes, containing a 16-bit binary value. The CRC value is calculated by the transmitting device, which appends the CRC to the message. The receiving device recalculates a CRC during reception of the message, and compares the calculated value to the actual value it received in the CRC field. If the two values are not equal, an error results.

The CRC is started by first preloading a 16-bit register to all 1's. Then a process begins of applying successive 8-bit bytes of the message to the current contents of the register. Only the eight bits of data in each character are used for generating the CRC. Start and stop bits and the parity bit do not apply to the CRC.

During generation of the CRC, each 8-bit character is exclusive ORed with the register contents. Then the result is shifted in the direction of the least significant bit (LSB), with a zero filled into the most significant bit (MSB) position. The LSB is extracted and examined. If the LSB was a 1, the register is then exclusive ORed with a preset, fixed value. If the LSB was a 0, no exclusive OR takes place. This process is repeated until eight shifts have been performed. After the last (eighth) shift, the next 8-bit byte is exclusive ORed with the register's current value, and the process repeats for eight more shifts as described above.

The final contents of the register, after all the bytes of the message have been applied, is the CRC value. When the CRC is appended to the message, the low-order byte is appended first, followed by the high-order byte.

14.11 IEEE 488 Standard

The Hewlett-Packard interface bus (HP-IB) was developed to interface smart test instruments such as digital voltmeter, signal generators, printers, display devices, etc. to the computer. In 1975 it was adopted as an IEEE standard and in 1978 it was revised to its present form and is now known as IEEE 488 standard. It is often referred to as the General Purpose Interface Bus (GPIB).

The standard defines three types of devices that may be connected on the GPIB : Talkers, Listeners, and controllers.

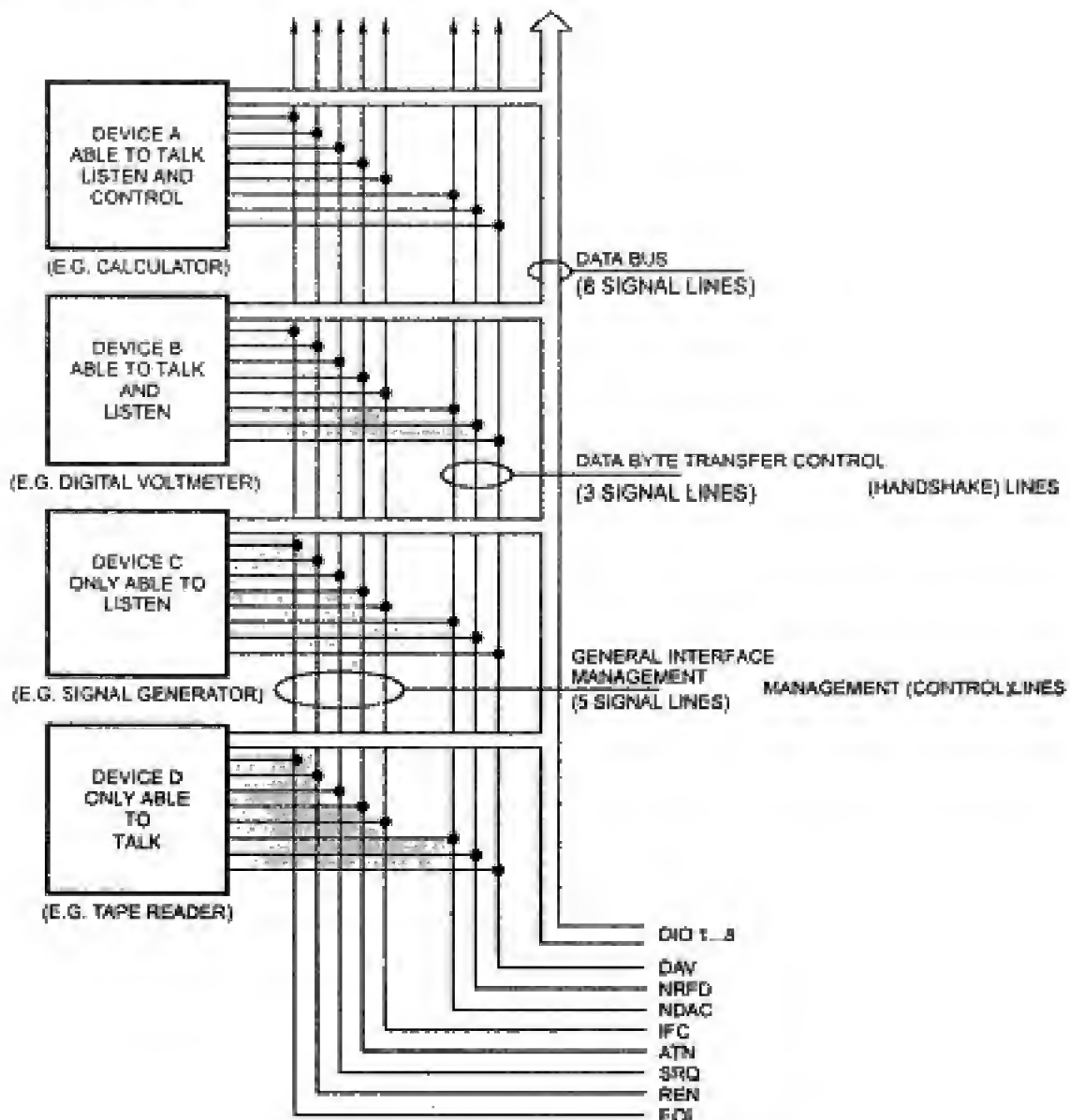


Fig. 14.26 Bus structure of the GPIB

1. Talkers are devices capable of putting data onto the bus for transmission to listeners or the controller in charge. Examples of talkers are tape readers, digital voltmeters, frequency counters, and other measuring instruments.
2. Listeners are devices capable of reading data on the bus output by talkers or the controller. Examples of listeners are printers, display devices, etc.
3. Controllers are devices which determine who talks and who listens on the bus. The controllers also program devices on the bus.

Note : A device can be both talker and listener.

Fig. 14.26 illustrates the bus structure of the GPIB. It has eight bidirectional data lines supported by three handshake control signals and five-general purpose interface management lines. The data lines are used to transfer data, addresses, commands, and status bytes among as many as 8 to 10 instruments. Fig. 14.27 shows the formats for the combination command address codes that a controller can send to talker and listeners. Bit 8 of these words is a don't care, bits 7 and 6 specify which command is being sent, and bits 1 to 5 give the address of the talker or listener to which the command is to be sent.

| CODE | | | | | | | | MEANING |
|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|--------------------|
| D ₈ | D ₇ | D ₆ | D ₅ | D ₄ | D ₃ | D ₂ | D ₁ | |
| X | 0 | 0 | 0 | B ₄ | B ₃ | B ₂ | B ₁ | UNIVERSAL COMMANDS |
| X | 0 | 1 | A ₅ | A ₄ | A ₃ | A ₂ | A ₁ | LISTEN ADDRESSES |
| X | 0 | 1 | 1 | 1 | 1 | 1 | 1 | UNLISTEN COMMAND |
| X | 1 | 0 | A ₅ | A ₄ | A ₃ | A ₂ | A ₁ | TALK ADDRESSES |
| X | 1 | 0 | 1 | 1 | 1 | 1 | 1 | UNTALK COMMAND |
| X | 1 | 1 | A ₅ | A ₄ | A ₃ | A ₂ | A ₁ | SECONDARY COMMANDS |
| X | 1 | 1 | 1 | 1 | 1 | 1 | 1 | IGNORED |

CODE FOR TYPE OF COMMAND

NOTES : THESE CODES ARE ONLY VALID WHEN ATN IS LOW.

ADDRESS 11111 CANNOT BE USED FOR A LISTENER OR A TALKER

Fig. 14.27 Formats for the combination command address codes

For example, to enable a device at address 05 as a talker, the controller simply asserts the ATN line low and sends out a command address byte of X1000101 on the data bus.

The five management lines of GPIB function as follows :

1. **Interface clear (IFC) :** The interface clear line, when asserted by the controller, resets all devices on the bus to a initial state.
2. **Attention line (ATN) :** The attention line, when asserted (low) indicates that the controller is putting a universal command or an address-command such as "listen" on the data bus. When ATN line is high, the data bus contains data or a status byte.
3. **Service Request (SRQ) :** Service Request is similar to an interrupt. Any device that needs to transfer data on the bus asserts the SRQ line low. The controller then polls all the devices to determine which one needs service.
4. **Remote Enable (REN) :** By asserting remote enable signal, controller can directly control instruments rather than by its front panel switches.

5. End or Identify (EOI) : The end or identify signal is asserted by a talker to indicate that the transfer of a block of data is complete.

The three handshake signals data valid (DAV), not ready for data (NRFD), and not data accepted (NDAC) are used to co-ordinate the transfer of data bytes on the data bus.

Upon Power on reset following sequence of operations are performed.

1. The controller takes control of the bus and sends out an IFC signal to set all instruments on the to a known state.
2. The controller then sends out a series of commands with the ATN line asserts low.
3. The controller then periodically checks the SRQ line for a service request. If the SRQ line is low, the controller polls each device on the bus one after another or all at once until it finds the device requesting service.
4. When the controller determines the source of the SRQ, it asserts the ATN line low and sends listener address commands to each listener that is to receive the data and a talk address command to the talker that requested service.
5. The controller then raises the ATN line high, and data is transferred directly from the talker to the listeners using a double-handshake-signal sequence.

Fig. 14.28 shows the sequence of signals on the handshake lines for a transfer of data from a talker to several listeners. The talker asserts the DAV line low to indicate that a valid data byte is on the bus. The addressed listeners then pull NRFD low and start accepting the data. The DAV, NRFD, and NDAC lines are all open-collector. Therefore, any listener can hold NRFD low to indicate that it is not ready for data or hold NDAC low to indicate that it has not accepted data byte. Due to common NDAC and NRFD signals the data transfer speed depends on the slower listener. When the slowest device accepts data byte, NRFD line is released high. The talker sense NRFD becoming high and asserts its DAV signal low, with data on data bus. The listeners pull NDAC low again, and sequence is repeated until the talker has sent all the data bytes it has to send.

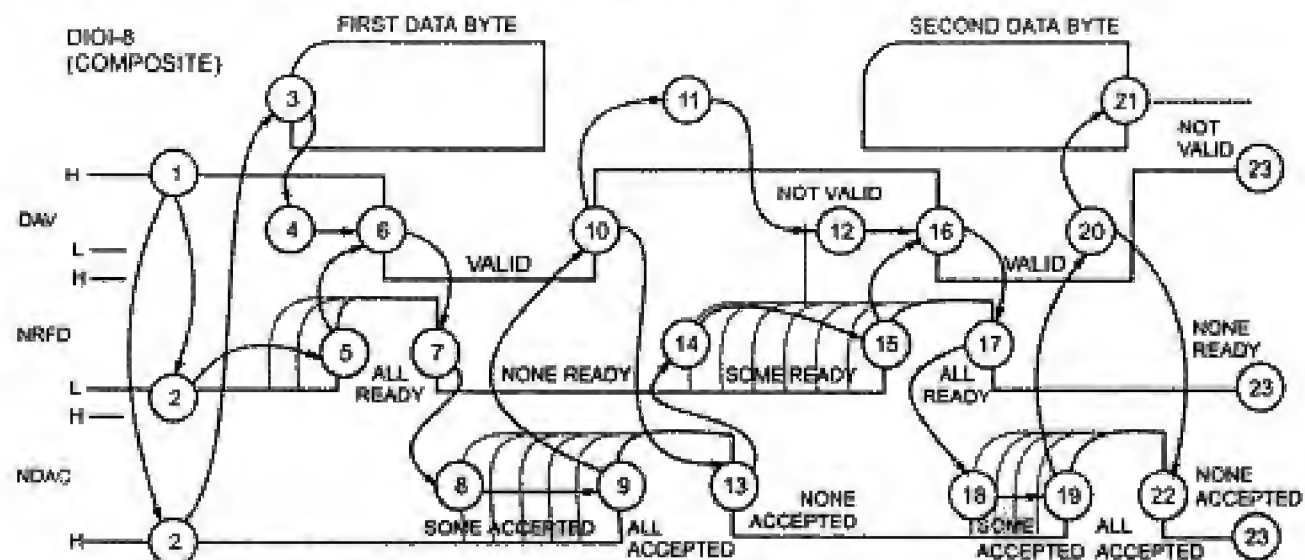


Fig. 14.28 Sequence of signals on the handshake lines

Fig. 14.29 shows the 24 pin GPIB connector. Each of the eight control signals has a separate ground pin (EOI and REN share pin 24) and a shield or frame ground is also provided. These separate grounds and shielding is provided for noise immunity.

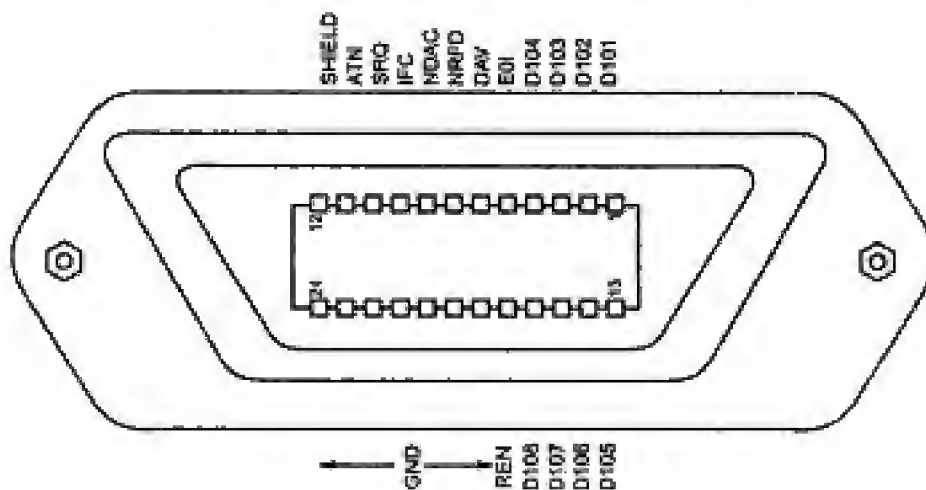


Fig. 14.29 GPIB connector

Electrical characteristics of GPIB :

- 1) Voltage levels : TTL compatible.
- 2) Driver requirements : $V_{OL} \leq 0.5 \text{ V}$ at 48 mA sink current
and $V_{OH} \geq 2.4 \text{ V}$ at 5.2 mA source current
- 3) Receiver requirements : $V_{IH} \leq 2.0 \text{ V}$
 $V_{IL} \geq 0.8 \text{ V}$
- 4) Noise immunity : 0.3 V in low state
0.4 V in high state
- 5) Data transfer rate : 1 MB/sec for cable less than 15 m in length.

Review Questions

1. Classify serial data transmission.
2. Explain asynchronous serial communication.
3. Explain synchronous serial communication.
4. Give pin description of RS-232C DB-25P connector.
5. Explain DB-9P connector.
6. How data is transmitted and received using RS-232C ?
7. Explain interconnection between RS-232C and microcontroller.
8. Write short note on
 - 1) RS-423-A
 - 2) RS-422-A
 - 3) RS-485
9. Compare different types of EIA standards.

10. What are the features of I^2C bus ?
11. Give bus configuration of I^2C bus.
12. How data is transferred by I^2C bus ? Explain with timing diagram.
13. Give different types of address formats of I^2C bus.
14. What are different modes of I^2C bus ? Explain any one of them in detail.
15. Write a note on i) I^2C bus ii) SPI
16. Explain clock polarity and clock phase in SPI.
17. What are the advantages of SPI over I^2C ?
18. Explain in detail different modes in modulus protocols.
19. Explain modulus message framing.
20. With respect to Modbus explain error checking.
21. Write a note on IEEE 488 standard.



Interfacing to EEPROM and RTC DS1307

15.1 Serial EEPROMs

Serial EEPROMs are small Electrically Erasable Programmable Read Only Memory chips. These devices are usually used to store user configurable parameters and device serial numbers. They use a serial bus interface, which allows them to be packed in inexpensive 8 pin packages.

It allows to store a small amount of data in non-volatile memory, using only a few of the port pins, and without raising the system's cost much. They are usually specified to retain the data for 10 years and to endure 100,000 write operations before failure. They only require a 5 volt power supply (some 3 V versions also exist).

Because these chips use a serial interface, they cannot be read quickly enough to serve as conventional memory. In addition, a considerable length of time (milliseconds) is required to perform a write operation. They typically hold less than 1024 bytes of memory.

15.2 Types of Serial EEPROM Chips

There are several types of Serial EEPROMs, but most of them fall into either a 2-wire or 3-wire interface category. Usually, the 3-wire devices require an addition wire (beyond the 3 for data transfer) for each chip to be used. The 2-wire interface, called I^2C or IIC or "I square C" uses only two wires, regardless of how many chips are attached. I^2C is a trademark of Philips. The three wire interfaces include SPI and Microwire, which is a trademark of National Semiconductor.

15.3 EEPROM : 93C46/56/66

The 93C46/56/66 provides 1024/2048/4096 bits of serial electrically erasable programmable read-only memory (EEPROM), organized as 64/128/256 of 16 bits each (when the ORG pin is connected to V_{CC}), and 128/256/512 words of 8 bits each (when the ORG pin is tied to ground). The device is optimized for use in many industrial and commercial applications where low-power and low-voltage operations are essential. The 93C46/56/66 is available in space-saving 8-lead PDIP, 8-lead JEDEC SOIC, 8-lead EIAJ SOIC, 8-lead MAP, 8-lead TSSOP, and 8-lead dBG2 packages, as shown in the Fig. 15.1.

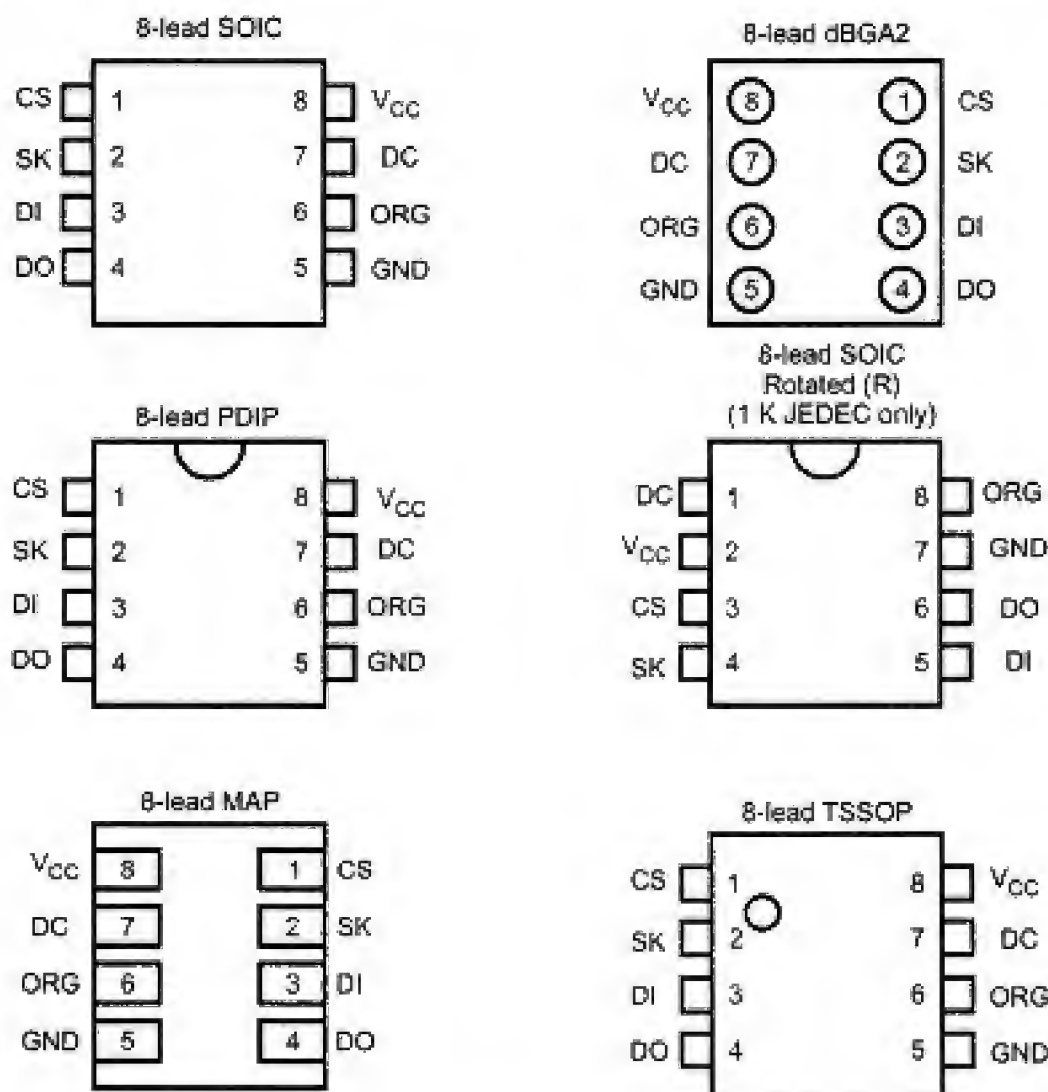


Fig. 15.1

The 93C46/56/66 is enabled through the Chip Select pin (CS) and accessed via a three-wire serial interface consisting of Data Input (DI), Data Output (DO), and Shift Clock (SK). Upon receiving a Read instruction at DI, the address is decoded and the data is clocked out serially on the DO pin. The Write cycle is completely self-timed, and no separate Erase cycle is required before Write. The Write cycle is only enabled when the part is in the Erase/Write Enable state. When CS is brought high following the initiation of a Write cycle, the DO pin outputs the Ready/Busy status of the part.

The Table 15.1 shows the pin configuration of 93C46/56/66 EEPROM.

Pin configurations

| Pin Name | Function |
|-----------------|-----------------------|
| CS | Chip Select |
| SK | Serial Data Clock |
| DI | Serial Data Input |
| DO | Serial Data Output |
| GND | Ground |
| V _{CC} | Power Supply |
| ORG | Internal Organization |
| DC | Don't Connect |

Table 15.1

The Fig. 15.2 shows the block diagram of 93CXX EEPROM.

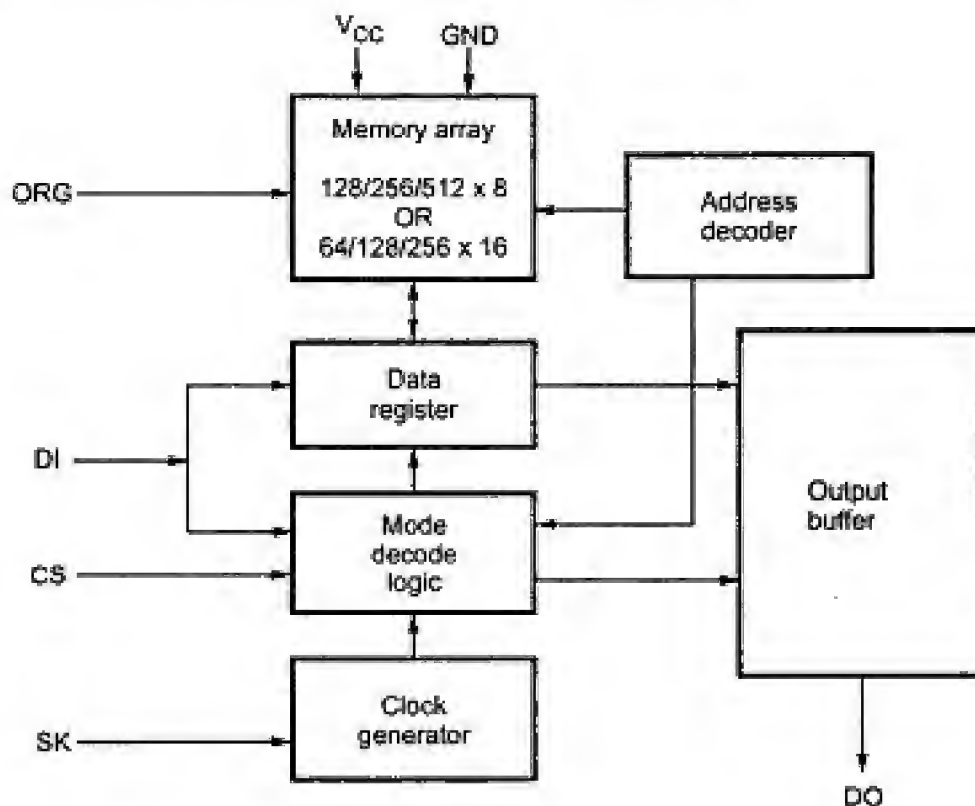


Fig. 15.2 Block diagram

Functional Description

The 93C46/56/66 is accessed via a simple and versatile three-wire serial communication interface. Device operation is controlled by seven instructions issued by the host processor. A valid instruction starts with a rising edge of CS and consists of a start bit (logic "1") followed by the appropriate opcode and the desired memory address location.

READ (READ) : The Read (READ) instruction contains the address code for the memory location to be read. After the instruction and address are decoded, data from the selected memory location is available at the serial output pin DO. Output data changes are synchronized with the rising edges of serial clock SK. It is important to note that a dummy bit (logic "0") precedes the 8 or 16-bit data output string.

ERASE (ERASE) : The Erase (ERASE) instruction programs all bits in the specified memory location to the logical "1" state. A logic "1" at pin DO indicates that the selected memory location has been erased and the part is ready for another instruction.

WRITE (WRITE) : The Write (WRITE) instruction contains the 8 or 16-bits of data to be written into the specified memory location. A logic "0" at DO indicates that programming is still in progress. A logic "1" indicates that the memory location at the specified address has been written with the data pattern contained in the instruction and the part is ready for further instructions.

ERASE ALL (ERAL) : The Erase All (ERAL) instruction programs every bit in the memory array to the logic "1" state.

WRITE ALL (WRAL) : The Write All (WRAL) instruction programs all memory locations with the data patterns specified in the instruction.

ERASE/WRITE DISABLE (EWDS) : To protect against accidental data disturb, the Erase/Write Disable (EWDS) instruction disables all programming modes and should be executed after all programming operations. The operation of the Read instruction is independent of both the EWEN and EWDS instructions and can be executed at any time.

93CXX Serial EEPROM Interface

The 93CXX may be connected to the 89CXX051 microcontroller in either a 3-wire or 4-wire configuration as shown in the Fig. 15.3. In the 3-wire configuration, the EEPROM serial data in (DI) and serial data out (DO) pins are both connected to the same microcontroller I/O pin, thereby saving a pin. This is possible because the microcontroller I/O pins can be dynamically reprogrammed as input or output.

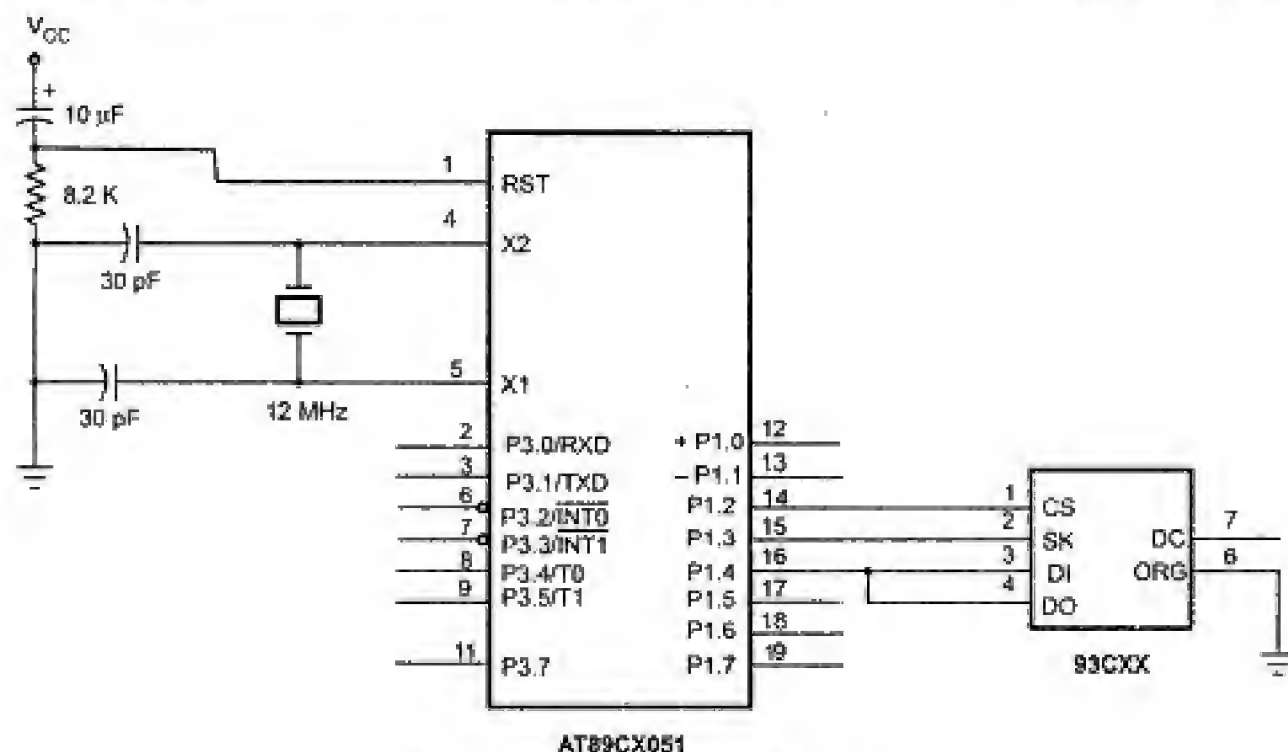


Fig. 15.3 (a) 3-Wire configuration

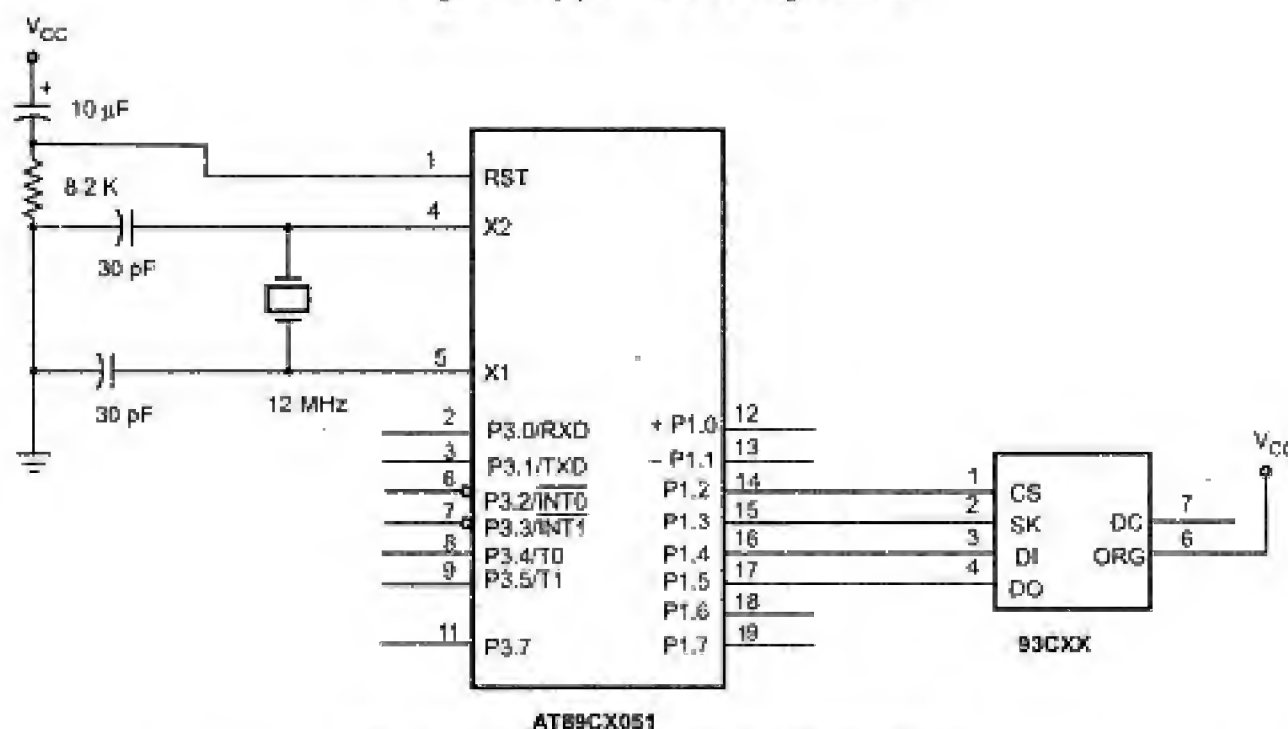


Fig. 15.3 (b) Typical circuit configuration

Let us see the connections of the 93CXX ORG pins shown in Fig. 15.3 (a) and Fig. 15.3 (b). The ORG (internal organization) pin selects 8-bit data when grounded and 16-bit data when floating or tied to V_{CC} . The ORG pin connections shown in the figures are for illustration only; 8-bit or 16-bit data may be selected in either the 3-wire or 4-wire configuration.

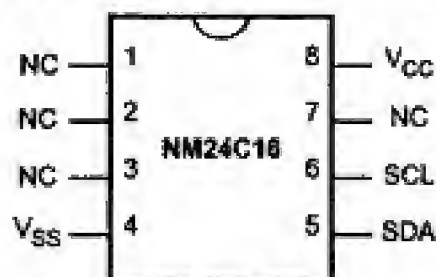
15.4 EEPROM : 24C16

The 24C16 device is 16,384 bits of CMOS non-volatile electrically erasable memory. This device conforms to all specifications in the Standard I²C 2-wire protocol and are designed to minimize device pin count, and simplify PC board layout requirements.

This communications protocol uses CLOCK (SCL) and DATA I/O (SDA) lines to synchronously clock data between the master (for example a microcontroller) and the slave EEPROM device(s).

The Fig. 15.4 shows pin diagram for 24C16 EEPROM.

Dual-in-line package (N) and SO package (M8)



TSSOP package (MT8)



Fig. 15.4

The Table 15.2 gives the pin description for 24C16 EEPROM.

| | |
|-----------------|--------------------|
| V _{SS} | Ground |
| SDA | Serial Data I/O |
| SCL | Serial Clock Input |
| NC | No Connection |
| V _{CC} | Power Supply |

Table 15.2

The Fig. 15.5 shows the block diagram of 24C16 EEPROM.

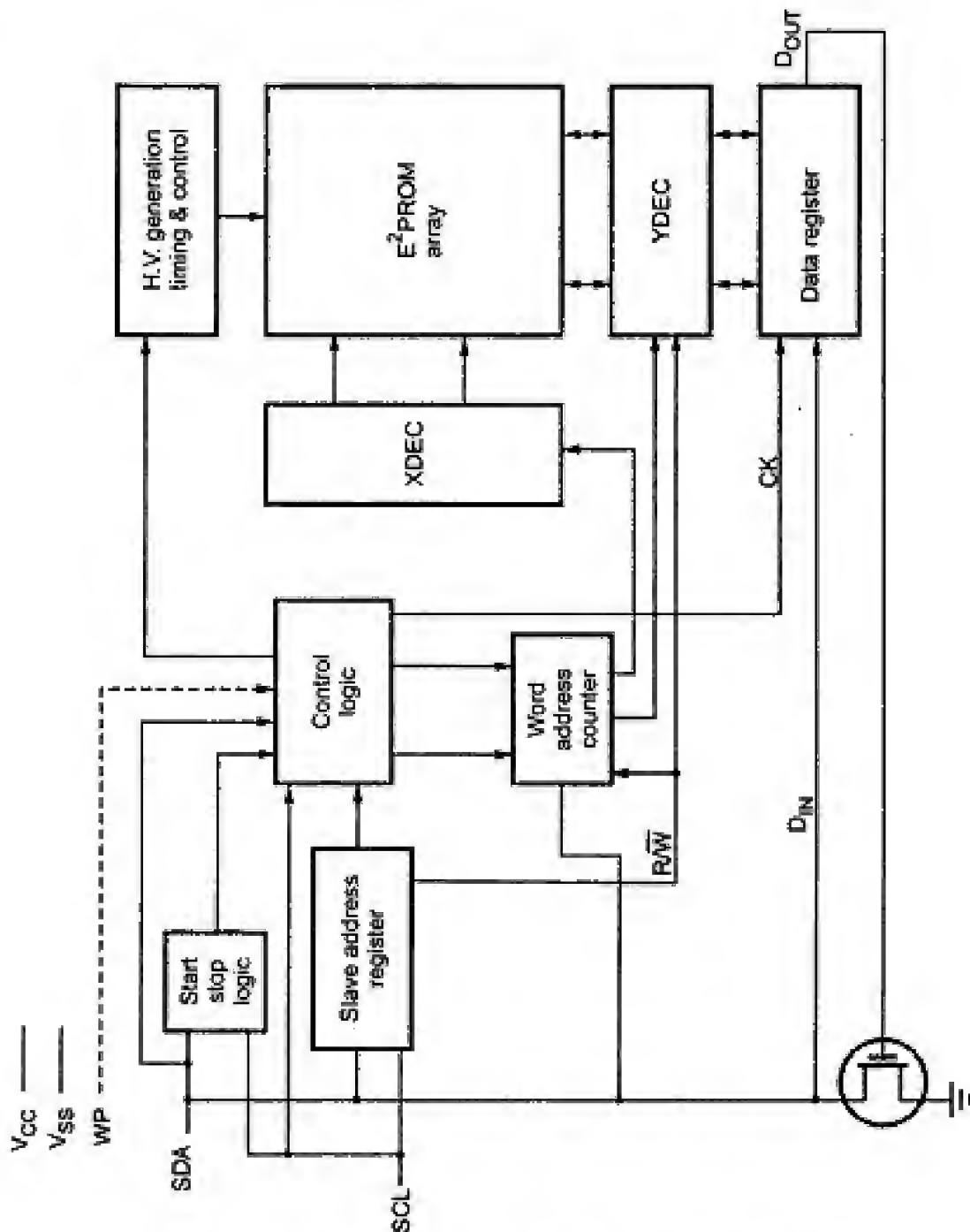


Fig. 15.5 Block diagram of 24C16 EEPROM

15.5 EEPROM : 24C32/64

The 24C16/32/64 provides 16K/32K/64K bits of serial electrically erasable and programmable read only memory (EEPROM) organized as 4096/8192 words of 8-bits each. The device's cascadable feature allows upto 8 devices to share a common 2-wire bus.

The 24C 16/32/64 is available in space saving 8-pin JEDEC PDIP, 8-pin JEDEC SOIC, 8-pin EIAJ SOIC, and 8-pin TSSOP (AT24C64) packages and is accessed via a 2-wire serial interface.

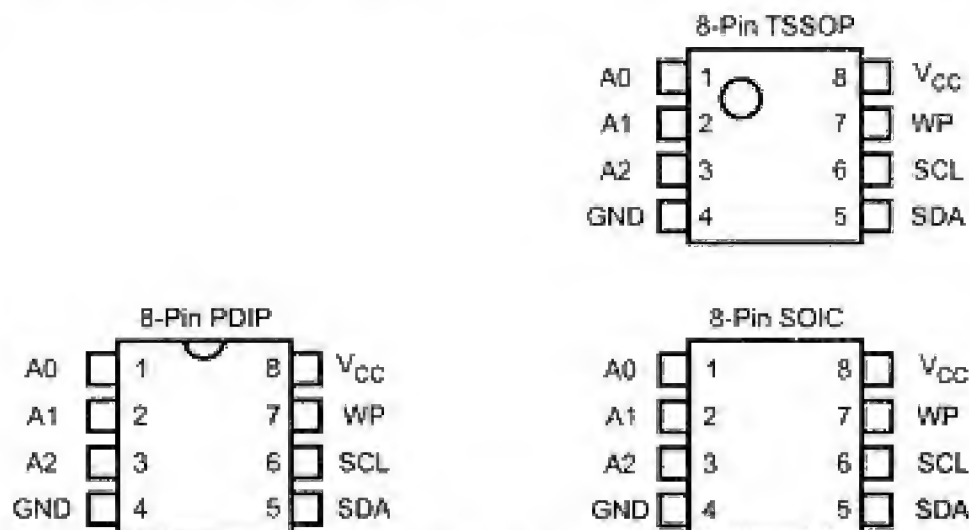


Fig. 15.6

The Table 15.3 shows the pin configuration for 24C 16/32/64 EEPROM.

| Pin Name | Function |
|----------|--------------------|
| A0 - A2 | Address Inputs |
| SDA | Serial Data |
| SCL | Serial Clock Input |
| WP | Write Protect |

Table 15.3

Pin Description

Serial Clock (SCL) : The SCL input is used to positive edge clock data into each EEPROM device and negative edge clock data out of each device.

Serial Data (SDA) : The SDA pin is bidirectional for serial data transfer. This pin is open-drain driven and may be wire-ORed with any number of other open-drain or open collector devices.

Device/Page Addresses (A2, A1, A0) : The A2, A1 and A0 pins are device address inputs that are hard wired or left not connected for hardware compatibility with AT24CXX. When the pins are hardwired, as many as eight 32 K/64 K devices may be

addressed on a single bus system. When the pins are not hardwired, the default A_2 , A_1 and A_0 are zero.

Write Protect (WP) : The write protect input, when tied to GND, allows normal write operations. When WP is tied high to V_{CC} , all write operations to the upper quadrant (8/16 K bits) of memory are inhibited. If left unconnected, WP is internally pulled down to GND.

The Fig. 15.7 shows the block diagram for 24C32/64 EEPROM.

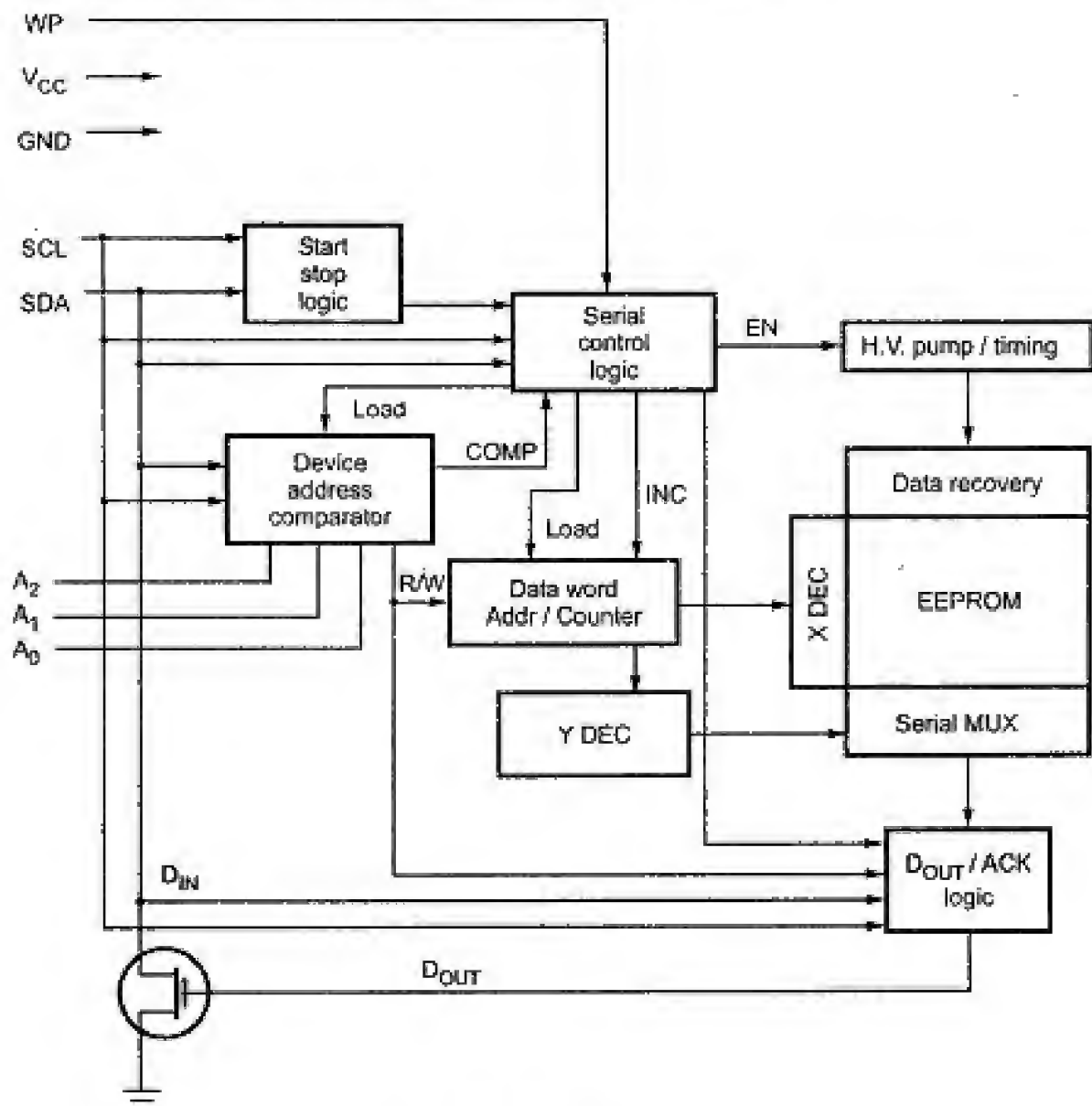


Fig. 15.7

24CXX Serial EEPROM Interface

The Fig. 15.8 shows the 24CXX interface to 8051 microcontroller. The 24CXX EEPROM uses I^2C interface and I^2C family of devices interfaces directly with Intel 8051.

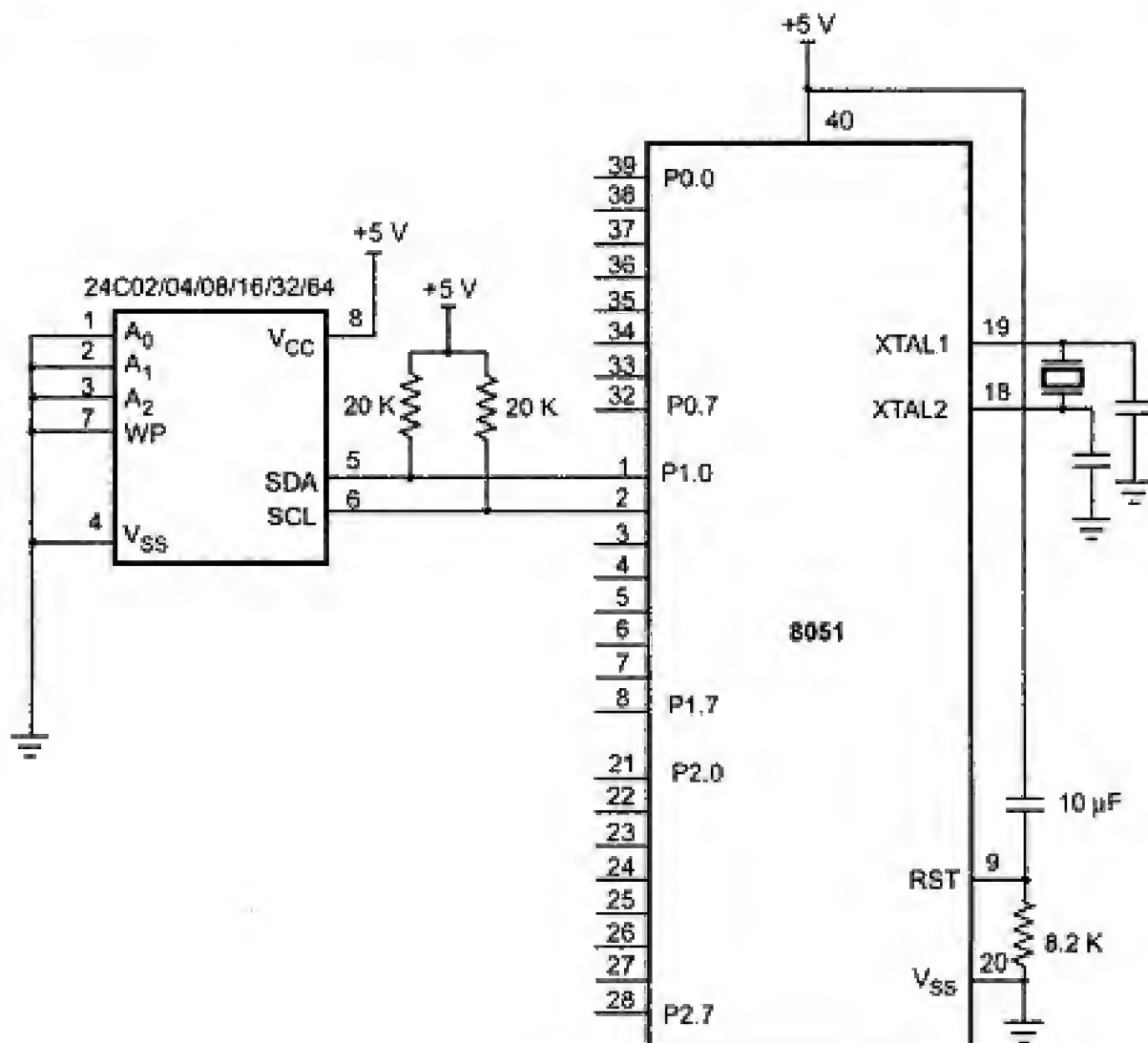


Fig. 15.8

I²C E²PROMs are 2-wire interface, nonvolatile memories ranging from 2K bits (24C02) to 64K bits (24C64) in density. They adhere to the I²C protocol which uses 2 lines, a data (SDA) and serial clock (SCL) line for all transmissions, as described in Chapter 14.

The 24C02 E²PROM has an 8 byte page write buffer and a write protect pin for inadvertent write protection. The 24C04, 24C08 and 24C16 devices have 16 byte page write buffers. Up to eight 24C02 devices, four 24C04 devices, two 24C08 devices and one 24C16 device may be connected to an I²C bus and addressed independently. The 24C32/64 has a 32-byte Page Write buffer and up to eight devices may be connected to an I²C bus and addressed independently. Unique addressing is accomplished through hard-wiring address pins A₀, A₁ and A₂ on each device.

15.6 RTC DS1307

The DS1307 Serial Real Time Clock is a low-power, full BCD clock/calendar plus 56 bytes of nonvolatile SRAM. In DS1307 address and data are transferred serially via the 2-wire bi-directional bus. The clock/calendar of DS1307 provides seconds, minutes, hours, day, date, month, and year information. The end of the month date is automatically adjusted for months with less than 31 days, including corrections for leap year. The clock operates in either the 24-hour or 12-hour format with AM/PM indicator. The DS1307 has a built-in power sense circuit which detects power failures and automatically switches to the battery supply.

15.6.1 Features of DS1307

- Real time clock counts seconds, minutes, hours, date of month, month, day of week and year with leap year compensation valid up to 2100.
- 56 byte nonvolatile RAM for general data storage.
- 2-wire interface (I²C) compatible.
- Automatic power fail detect.
- Consumes less than 500 nA in battery back-up mode.
- Optional Industrial temperature range : - 40° C to + 35° C
- Available in 8-pin plastic DIP or SO

The Fig. 15.9 shows the pin diagram of DS1307 RTC

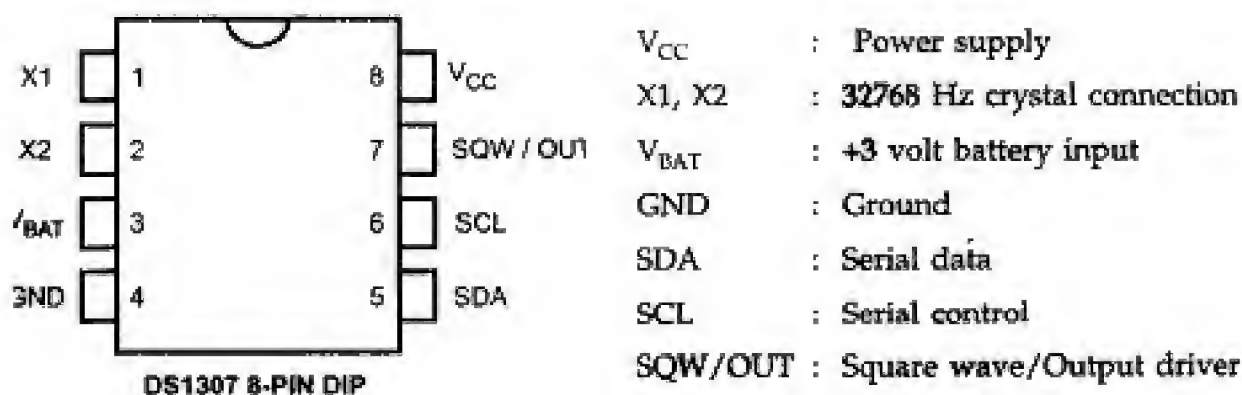


Fig. 15.9

15.6.2 DS1307 Operation

The DS1307 operates as a slave device on the serial bus. Access is obtained by implementing a START condition and providing a device identification code followed by a register address. Subsequent registers can be accessed sequentially until a STOP condition is executed.

15.6.3 DS1307 Block Diagram

The Fig. 15.10 shows the block diagram of RTC DS1307.

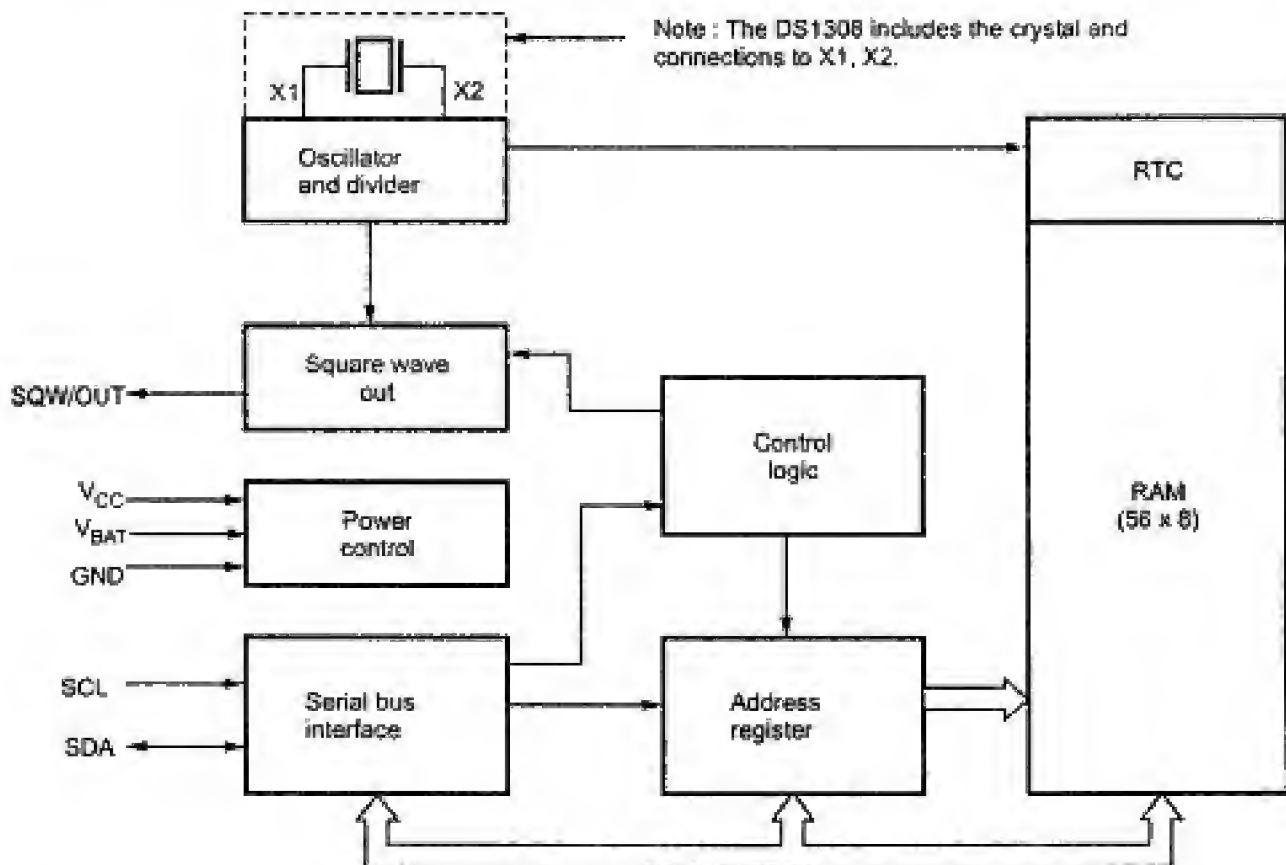


Fig. 15.10 Block diagram of RTC DS1307

15.6.4 Signal Descriptions

V_{CC}, GND - DC power is provided to the device on these pins. V_{CC} is the +5 volt input. When 5 volts is applied within normal limits, the device is fully accessible and data can be written and read. When a 3 volt battery is connected to the device and V_{CC} is below $1.25 \times V_{BAT}$, reads and writes are inhibited. However, the Timekeeping function continues unaffected by the lower input voltage. As V_{CC} falls below V_{BAT} the RAM and timekeeper are switched over to the external power supply (nominal 3.0 V DC) at V_{BAT}.

V_{BAT} : Battery input for any standard 3 volt lithium cell or other energy source. Battery voltage must be held between 2.0 and 3.5 volts for proper operation.

SCL (Serial Clock Input) : SCL is used to synchronize data movement on the serial interface.

SDA (Serial Data Input/Output) : SDA is the input/output pin for the 2-wire serial interface. The SDA pin is open drain which requires an external pullup resistor.

SQW/OUT (Square Wave/Output Driver) : When enabled, the SQWE bit set to 1, the SQW/OUT pin outputs one of four square wave frequencies (1 Hz, 4 kHz, 8 kHz, 32 kHz). The SQW/OUT pin is open drain which requires an external pullup resistor.

X1, X2 : Connections for a standard 32.768 kHz quartz crystal.

15.6.5 RTC and RAM Address Map

The address map for the RTC and RAM registers of the DS1307 is shown in Fig. 15.11. The real time clock registers are located in address locations 00h to 07h. The RAM registers are located in address locations 08h to 3Fh. During a multi-byte access, when the address pointer reaches 3Fh, the end of RAM space, it wraps around to location 00h, the beginning of the clock space.

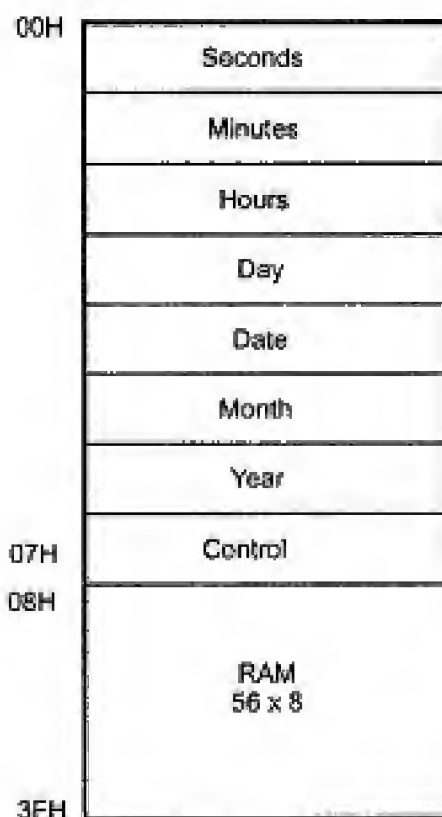


Fig. 15.11 DS1307 address map

15.6.6 Clock and Calender

The time and calender information is obtained by reading the appropriate register bytes. The real time clock registers are illustrated in Fig. 15.12. The time and calender are set or initialized by writing the appropriate register bytes. The contents of the time and calender registers are in the Binary Coded Decimal (BCD) format. Bit 7 of Registers 0 is the Clock Halt (CH) bit. When this bit is set to a 1, the oscillator is disabled. When cleared to a 0, the oscillator is enabled.

Please note that the initial power on state of all registers is not defined. Therefore it is important to enable the oscillator (CH bit = 0) during initial configuration.

The DS1307 can be run in either 12 hour or 24 hour mode. Bit 6 of the hours register is defined as the 12 or 24 hour mode select bit. When high, the 12 hour mode is selected. In the 12 hour mode, bit 5 is the AM/PM bit with logic high being PM. In the 24 hour mode, bit 5 is the second 10 hour bit (20-23 hours).

| | | | | | | | | |
|-----|---------|------------|-----------|----------|---------|-----|-----|----------------------------|
| | Bit 7 | | | | | | | Bit 0 |
| 00H | CH | 10 Seconds | | | Seconds | | | 00-59 |
| | X | 10 Minutes | | | Minutes | | | 00-59 |
| | X | 12 / 24 | 10 HR A/P | 10 HR | Hours | | | 01-12 00-23 |
| | X | X | X | X | X | Day | | 1-7 |
| | X | X | 10 Date | | Date | | | 01-28/29 01-30 01-31 |
| | X | X | X | 10 Month | Month | | | 01-12 |
| | 10 Year | | | | Year | | | 00-99 |
| | OUT | X | X | SQWE | X | X | RS1 | RS0 |
| 07H | | | | | | | | |

Fig. 15.12 DS1307 timekeeper registers

15.6.7 Control Register

The DS1307 Control Register is used to control the operation of the SQW/OUT pin.

| BIT 7 | BIT 6 | BIT 5 | BIT 4 | BIT 3 | BIT 2 | BIT 1 | BIT 0 |
|-------|-------|-------|-------|-------|-------|-------|-------|
| OUT | X | X | SQWE | X | X | RS1 | RS0 |

OUT (Output control) : This bit controls the output level of the SQW/OUT pin when the square wave output is disabled. If SQWE = 0, the logic level on the SQW/OUT pin is 1 if OUT = 1 and is 0 if OUT = 0.

SQWE (Square Wave Enable) : This bit, when set to a logic 1, will enable the oscillator output. The frequency of the square wave output depends upon the value of the RS0 and RS1 bits.

RS (Rate Select) : These bits control the frequency of the square wave output when the square wave output has been enabled. Table 15.4 lists the square wave frequencies that can be selected with the RS bits.

| RS1 | RS0 | SQW output frequency |
|-----|-----|----------------------|
| 0 | 0 | 1 Hz |
| 0 | 1 | 4.096 kHz |
| 1 | 0 | 8.192 kHz |
| 1 | 1 | 32.768 kHz |

Table 15.4 Square wave output frequency

15.6.8 2-Wire Serial Data Bus

The DS1307 supports a bi-directional 2-wire bus and data transmission protocol. A device that sends data onto the bus is defined as a transmitter and a device receiving data as a receiver. The device that controls the message is called a master. The devices that are controlled by the master are referred to as slaves. The bus must be controlled by a master device which generates the serial clock (SCL), controls the bus access, and generates the START and STOP conditions. The DS1307 operates as a slave on the 2-wire bus. A typical bus configuration using this 2-wire protocol is shown in Fig. 15.13.

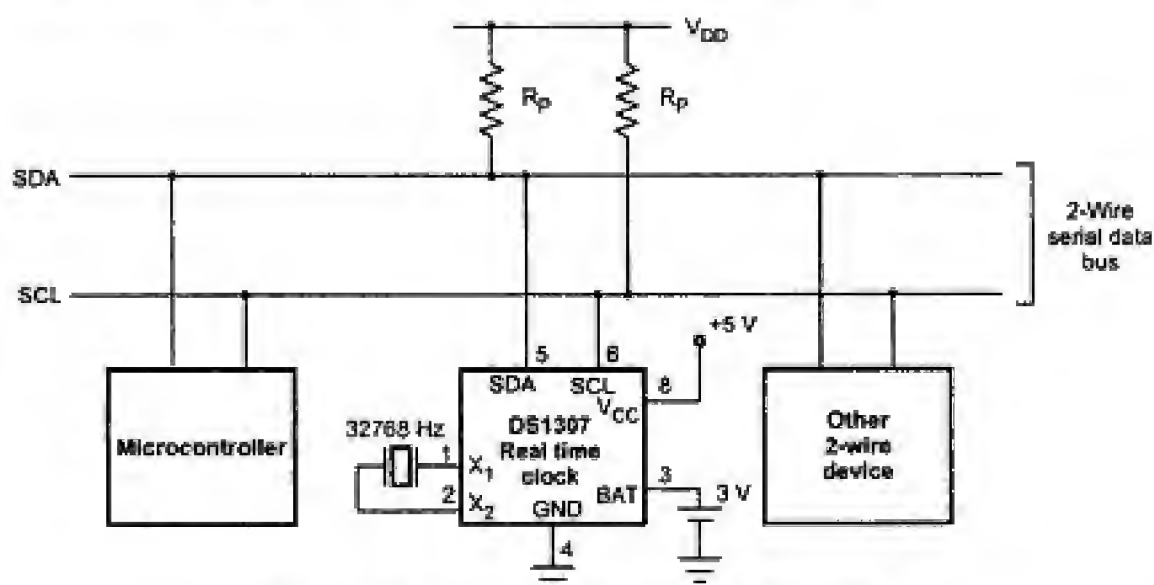


Fig. 15.13 Typical 2-wire bus configuration

The DS1307 has the 2-wire bus connected to two I/O port pins of the microcontroller : SCL - P1.0, SDA - P1.1. The V_{DD} voltage is 5 V, $R_p = 5\text{ k}\Omega$ and the microcontroller is using a 12 MHz crystal. The other peripheral device could be any other device that recognizes the 2-wire protocol, such as the DS1621 Digital Thermometer and Thermostat. The interface with the microcontroller was accomplished using the microcontroller Kit hardware and software.

During data transfer, the data line must remain stable whenever the clock line is high. Changes in the data line while the clock line is high will be interpreted as control signals. Accordingly, the following bus conditions have been defined :

Start data transfer : A change in the state of the data line from high to low, while the clock line is high, defines a START condition.

Stop data transfer : A change in the state of the data line from low to high, while the clock line is high, defines the STOP condition.

Data valid : The state of the data line represents valid data when, after a START condition, the data line is stable for the duration of the high period of the clock signal. The data on the line must be changed during the low period of the clock signal. There is one clock pulse per bit of data.

Each data transfer is initiated with a START condition and terminated with a STOP condition. The number of data bytes transferred between the START and the STOP conditions is not limited, and is determined by the master device. The information is transferred byte-wise and each receiver acknowledges with a ninth bit.

Acknowledge : Each receiving device, when addressed, is obliged to generate an acknowledge after the reception of each byte. The master device must generate an extra clock pulse which is associated with this acknowledge bit.

A device that acknowledges must pull down the SDA line during the acknowledge clock pulse in such a way that the SDA line is stable low during the high period of the acknowledge related clock pulse. Of course, setup and hold times must be taken into account. A master must signal an end of data to the slave by not generating an acknowledge bit on the last byte that has been clocked out of the slave. In this case, the slave must leave the data line high to enable the master to generate the STOP condition.

Fig. 15.14 details how data transfer is accomplished on the 2-wire bus. Depending on the state of the R/active-low W bit, two types of data transfer are possible :

1. **Data transfer from a master transmitter to a slave receiver :** The first byte transmitted by the master is the slave address. Next follows a number of data bytes. The slave returns an acknowledge bit after each received byte. Data is transferred with the most significant bit (MSB) first.
2. **Data transfer from a slave transmitter to a master receiver :** The first byte (the slave address) is transmitted by the master. The slave then returns an acknowledge bit. This is followed by the slave transmitting a number of data bytes. The master returns an acknowledge bit after all received bytes other than the last byte. At the end of the last received byte, a not acknowledge is returned.

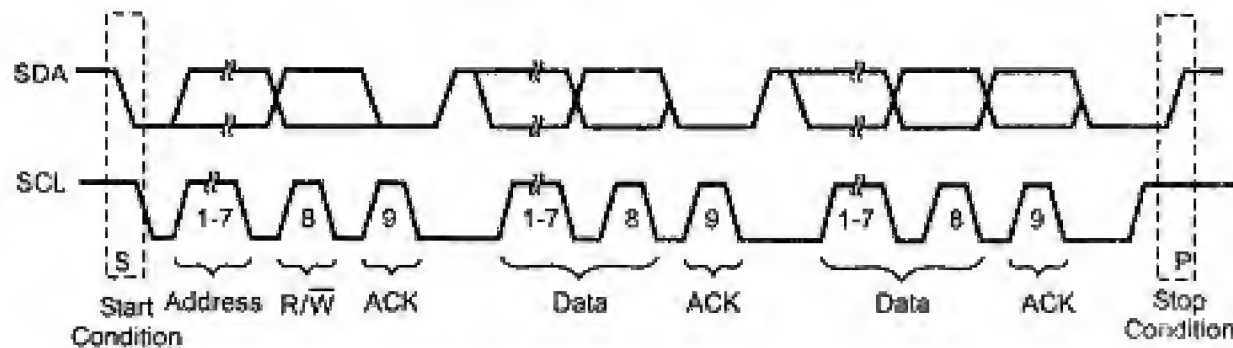


Fig. 15.14 Data transfer on 2-wire serial bus

The master device generates all of the serial clock pulses and the START and STOP conditions. A transfer is ended with a STOP condition or with a repeated START condition. Since a repeated START condition is also the beginning of the next serial transfer, the bus will not be released. Data is transferred with the most significant bit (MSB) first.

15.6.9 Operating Modes of DS1307

The DS1307 may operate in the following two modes:

1. **Slave receiver mode (DS1307 write mode)** : Serial data and clock are received through SDA and SCL. After each byte is received, an acknowledge bit is transmitted. START and STOP conditions are recognized as the beginning and end of a serial transfer. Address recognition is performed by hardware after reception of the slave address and direction bit as shown in Fig. 15.15. The address byte is the first byte received after the start condition is generated by the master. The address byte contains the 7-bit DS1307 address, which is 1101000, followed by the direction bit (R/active-low W) which for a write is a 0. After receiving and decoding the address byte, the DS1307 outputs an acknowledge on the SDA line. After the DS1307 acknowledges the slave address + write bit, the master transmits a register address to the DS1307. This will set the register pointer on the DS1307. The master will then begin transmitting each byte of data with the DS1307 acknowledging each byte received. The master will generate a stop condition to terminate the data write.

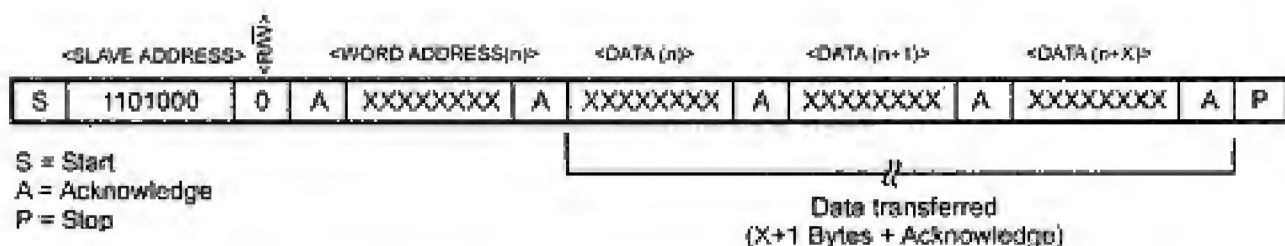


Fig. 15.15 Data write - slave receiver mode

2. **Slave transmitter mode (DS1307 read mode)** : The first byte is received and handled as in the slave receiver mode. However, in this mode, the direction bit will indicate that the transfer direction is reversed. Serial data is transmitted on SDA by the DS1307 while the serial clock is input on SCL. START and STOP conditions are recognized as the beginning and end of a serial transfer as shown in Fig. 15.16. The address byte is the first byte received after the start condition is generated by the master. The address byte contains the 7-bit DS1307 address, which is 1101000, followed by the direction bit (R/active-low W), which for a read is a 1. After receiving and decoding the address byte, the DS1307 inputs an acknowledge on the SDA line. The DS1307 then begins to transmit data starting with the register address pointed to by the register pointer. If the register pointer is not written to before the initiation of a read mode, the first address that is read is the last one stored in the register pointer. The DS1307 must be sent a Not-Acknowledge bit by the master to terminate a read.

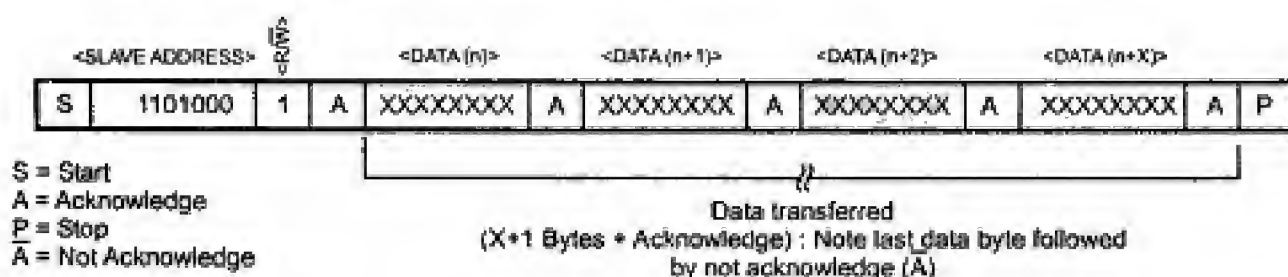


Fig. 15.16 Data read - slave transmitter mode

Review Questions

1. State the features of serial EEPROMs.
2. State the two types of interface used in serial EEPROMs.
3. List the features of 93CXX EEPROM.
4. Draw and explain the interface between 93CXX EEPROM and microcontroller AT89CX051.
5. Draw the block diagram of 93CXX EEPROM.
6. Explain the operation of 93CXX controlled by seven instructions issued by microcontroller.
7. Give the features of 24C16 EEPROM.
8. Give the features of 24C32/64 EEPROM.
9. Draw and explain 24CXX serial EEPROM interface to 8051 microcontroller.
10. List the features of RTC DS1307.
11. Draw and explain the typical 2-wire configuration used to interface DS1307 to the microcontroller.
12. Explain the operating modes of DS1307.

Conceptual Study of Derivatives of Microcontroller

16.1 Introduction

In this chapter we are going to study the advanced features provided by the 8051 family microcontrollers.

16.2 One-Time-Programmable (OTP) Feature

The chip is to be packaged by some material, which is called 'encapsulation' and the material used is called as 'encapsulant'. There are two types of packages used for the microcontrollers with EPROM control storage, ceramic packages and plastic packages. We are familiar with an EPROM in ceramic package. It has a small window built in for erasing the contents of it. EPROM microcontrollers are also available in plastic packages with no windows. These are referred to as one-time programmable (OTP) packages. The selection of the appropriate option of the package, can have a significant impact on the final application's cost, size and quality.

Plastic encapsulants use an epoxy plotting compound. This is injected around a chip after it has been wired to a lead frame. The lead frame becomes the pins on the package. The lead frame is wired to the chip via very thin aluminium wires which are ultrasonically bonded to both, the chip and the lead frame. After hardening of the encapsulant, the chip is protected from light, moisture and physical damage from the outside world.

Fig. 16.1 shows OTP plastic and windowed ceramic package.

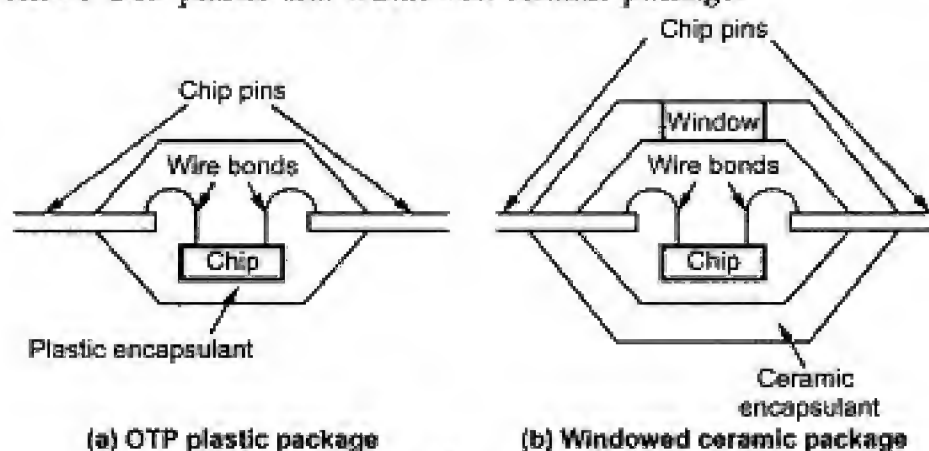


Fig. 16.1 Chip packaging
(16 - 1)

Now we will discuss the advantages and disadvantages of OTP packaging over ceramic packaging.

Advantages of OTP Packaging

1. The OTP packages are cheaper than the ceramic packages. The cost of windowed ceramic packages is 10 times more than that for plastic packages so ceramic packaging is only suitable for uses such as application debugging.
2. Using OTP parts is advantageous for the card manufacturer. If the different products are to be manufactured with the same device, then unprogrammed parts are kept in stock and the burning process is performed only for the required current build run. This is a real advantage in terms of inventory carrying costs. In this case, instead of keeping a number of products of same type, the manufacturer just needs to keep the minimum number of parts on hand to satisfy their current orders for the different products.
3. Most of the devices (ceramic packaging) are usually only available in a PTH (pin through hole) technology OTP (plastic devices) can be a very wide range.

Disadvantages of OTP Packaging

In OTP device, once the EPROM has been programmed, the device cannot be used for anything else. With windowed ceramic package, the EPROM can be erased and reprogrammed.

16.3 Features of 89C51 Microcontroller

Let us see the features of 89C51 microcontroller

- 4 kbytes of on-chip flash program memory
- Speed upto 33 MHz
- Fully static operation
- RAM expandable externally upto 64 kbytes
- 4 interrupt priority levels.
- 6 interrupt sources
- Four 8-bit I/O ports
- Full-duplex enhanced UART
 - Framing error detection
 - Automatic address recognition
- Three 16-bit timers/counters

- Power control modes
 - Clock can be stopped and resumed
 - Idle mode
 - Power down mode
- Programmable clock output
- Second DPTR register
- Asynchronous port reset
- Low EMI (inhibit ALE)
- Wake up from power down by an external interrupt

16.4 Microcontroller – 89C51 RD

16.4.1 Features and 89C51 RD

- On-chip Flash Program Memory upto 64 kbytes with In-System Programming (ISP) and In-Application Programming (IAP) capability. In-System Programming (ISP) allows the user to download new code while the microcontroller sits in the application. In-Application Programming (IAP) means that the microcontroller fetches new program code and reprograms itself while in the system. This allows for remote programming over a modem link.
- Boot ROM contains low level Flash programming routines for downloading via the UART
- Can be programmed by the end-user application (IAP)
- 6 clocks per machine cycle operation (standard)
- 12 clocks per machine cycle operation (optional)
- Speed up to 20 MHz with 6 clock cycles per machine cycle (40 MHz equivalent performance); up to 33 MHz with 12 clocks per machine cycle.
- Fully static operation
- RAM expandable externally to 64 kB
- 4 level priority interrupt
- 7 interrupt sources
- Four 8-bit I/O ports

- Full-duplex enhanced UART
 - Framing error detection
 - Automatic address recognition
- Power control modes
 - Clock can be stopped and resumed
 - Idle mode
 - Power down mode
- Programmable clock out
- Second DPTR register
- Asynchronous port reset
- Low EMI (inhibit ALE)
- Programmable Counter Array (PCA)
 - PWM
 - Capture/compare

16.4.2 Block Diagram

The Fig. 16.2 shows the block diagram of 80C51 family microprocessor and it shows added feature in bold letters.

16.4.3 Stop Clock Mode

This mode allows step-by-step utilization and permits reduced system power consumption by lowering the clock frequency down to any value.

16.4.4 Enhanced UART

In addition, the UART can perform framing error detection by looking for missing stop bits, and automatic address recognition. The UART also fully supports multiprocessor communication as does the standard 80C51 UART.

When used for framing error detection the UART looks for missing stop bits in the communication. A missing bit will set the FE bit in the SCON register. The FE bit shares the SCON.7 bit with SM0 and the function of SCON.7 is determined by PCON.6 (SMOD0) (see Fig. 16.3). If SMOD0 is set then SCON.7 functions as FE. SCON.7 functions as SM0 when SMOD0 is cleared. When used as FE SCON.7 can only be cleared by software. This is illustrated in Fig. 16.4.

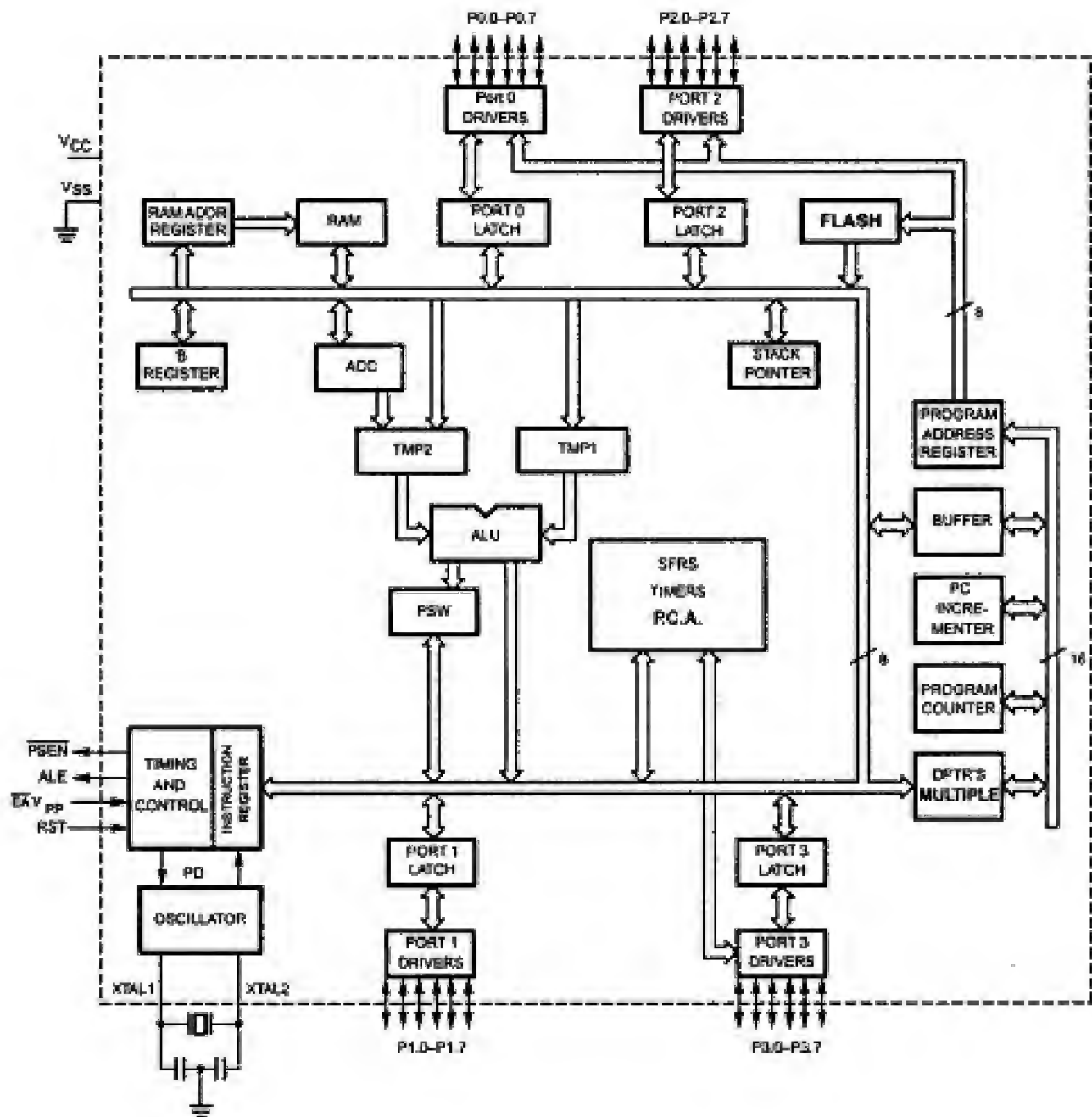


Fig. 16.2

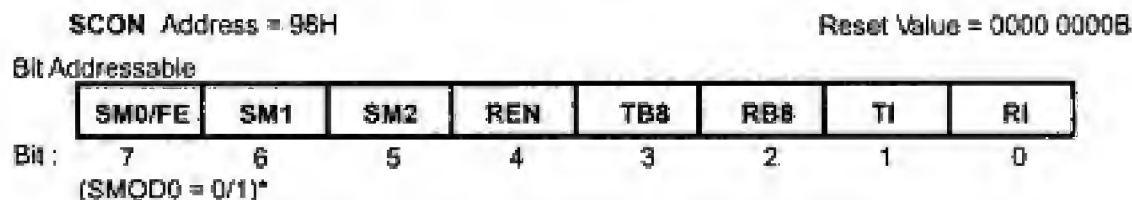


Fig. 16.3 SCON : Serial port control register

| Symbol | Function | | | |
|--------|---|------|----------------|---|
| FE | Framing Error bit. This bit is set by the receiver when an invalid stop bit is detected. The FE bit is not cleared by valid frames but should be cleared by software. The SMOD0 bit must be set to enable access to the FE bit. | | | |
| SM0 | Serial Port Mode Bit 0, (SMOD0 must = 0 to access bit SM0) | | | |
| SM1 | Serial Port Mode Bit 1 | | | |
| SM0 | SM1 | Mode | Description | Baud Rate** |
| 0 | 0 | 0 | shift register | $f_{osc}/6$ (6 clock mode) or $f_{osc}/12$ (12 clock mode) |
| 0 | 1 | 1 | 8-bit UART | variable |
| 1 | 0 | 2 | 9-bit UART | $f_{osc}/32$ or $f_{osc}/16$ (6 clock mode) or $f_{osc}/64$ or $f_{osc}/32$ (12 clock mode) |
| 1 | 1 | 3 | 9-bit UART | variable |
| SM2 | Enables the Automatic Address Recognition feature in Modes 2 or 3. If SM2 = 1 then RI will not be set unless the received 9th data bit (RB8) is 1, indicating an address, and the received byte is a Given or Broadcast Address. In Mode 1, if SM2 = 1 then RI will not be activated unless a valid stop bit was received, and the received byte is a Given or Broadcast Address. In Mode 0, SM2 should be 0. | | | |
| REN | Enables serial reception. Set by software to enable reception. Clear by software to disable reception. | | | |
| TB8 | The 9th data bit that will be transmitted in Modes 2 and 3. Set or clear by software as desired. | | | |
| RB8 | In modes 2 and 3, the 9th data bit that was received. In Mode 1, if SM2 = 0, RB8 is the stop bit that was received. In Mode 0, RB8 is not used. | | | |
| TI | Transmit interrupt flag. Set by hardware at the end of the 8th bit time in Mode 0, or at the beginning of the stop bit in the other modes, in any serial transmission. Must be cleared by software. | | | |
| RI | Receive interrupt flag. Set by hardware at the end of the 8th bit time in Mode 0, or halfway through the stop bit time in the other modes, in any serial reception (except see SM2). Must be cleared by software. | | | |

NOTE :

*SMOD0 is located at PCON6.

**fosc = oscillator frequency

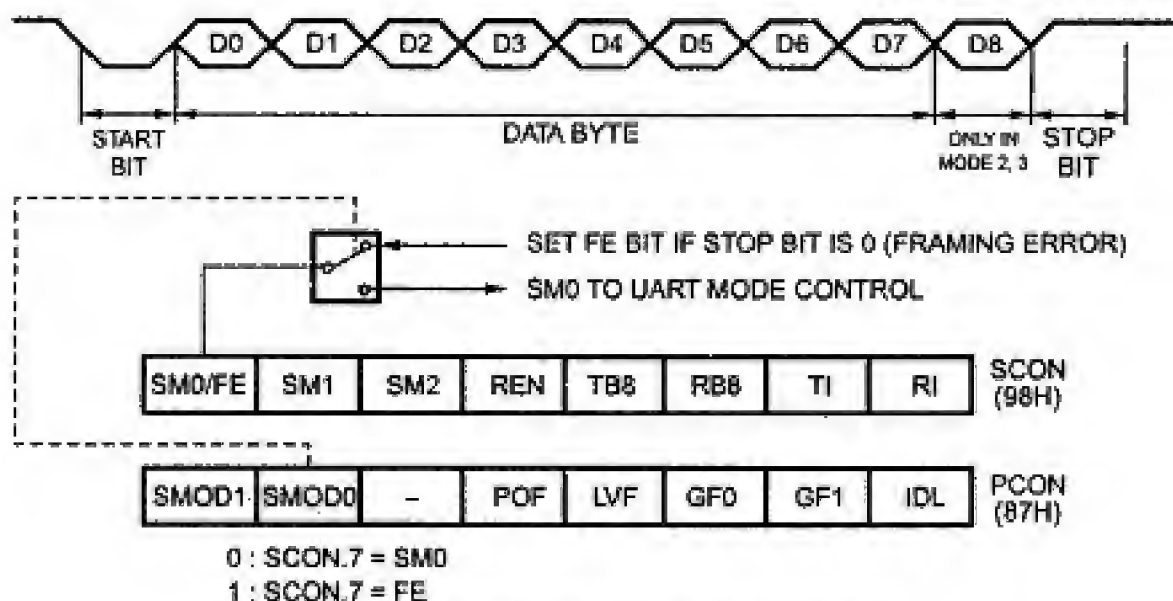


Fig. 16.4 UART framing error detection

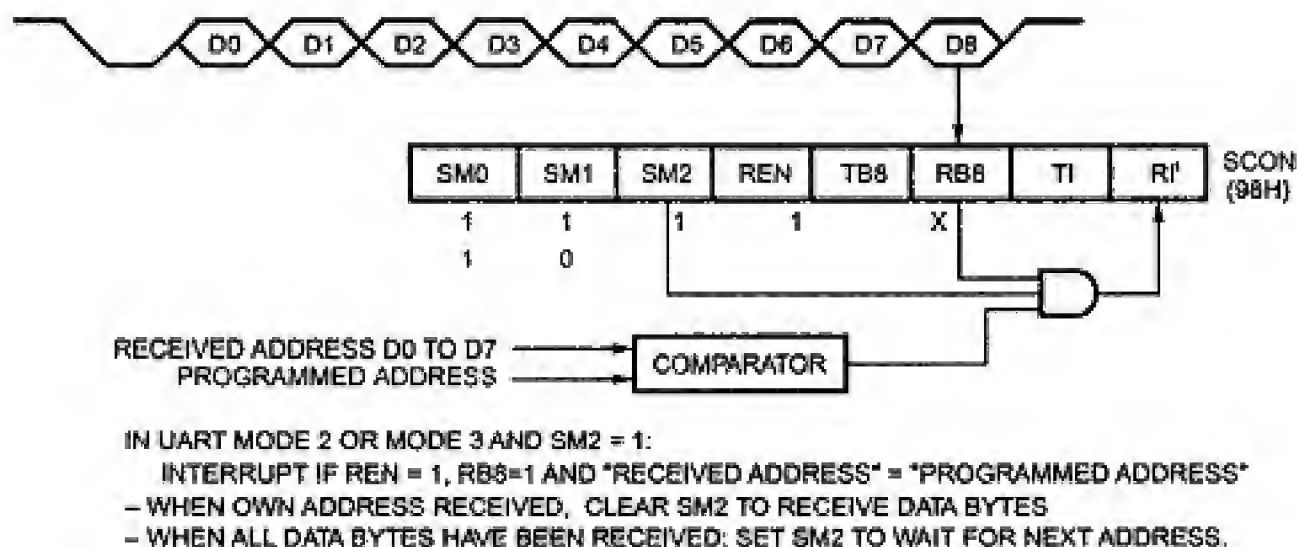


Fig. 16.5 UART Multiprocessor communication, automatic address recognition

16.4.5 Automatic Address Recognition

Automatic Address Recognition is a feature which allows the UART to recognize certain addresses in the serial bit stream by using hardware to make the comparisons. This feature saves a great deal of software overhead by eliminating the need for the software to examine every serial address which passes by the serial port. This feature is enabled by setting the SM2 bit in SCON. In the 9-bit UART modes, mode 2 and mode 3, the Receive Interrupt flag (RI) will be automatically set when the received byte contains either the "Given" address or the "Broadcast" address. The 9-bit mode requires that the 9th

information bit is 1 to indicate that the received information is an address and not data. Automatic address recognition is shown in Fig. 16.5.

Using the Automatic Address Recognition feature allows a master to selectively communicate with one or more slaves by invoking the Given slave address or addresses.

16.4.6 Dual DPTR

There are two 16-bit DPTR registers that address the external memory, and a single bit called DPS = AUXR1/bit 0 that allows the program code to switch between them.

16.4.7 Programmable Counter Array (PCA)

The Programmable Counter Array available on the 89C51RB2/RC2/RD2 is a special 16-bit Timer that has five 16-bit capture/compare modules associated with it. Each of the modules can be programmed to operate in one of four modes :

- Rising and/or falling edge capture
- Software timer
- High-speed output
- Pulse width modulator

Each module has a pin associated with it in port 1. Module 0 is connected to P1.3(CEX0), module 1 to P1.4(CEX1), etc. The basic PCA configuration is shown in Fig. 16.6.

The PCA timer is a common time base for all five modules and can be programmed to run at : 1/6 the oscillator frequency, 1/2 the oscillator frequency, the Timer 0 overflow, or the input on the ECI pin (P1.2).

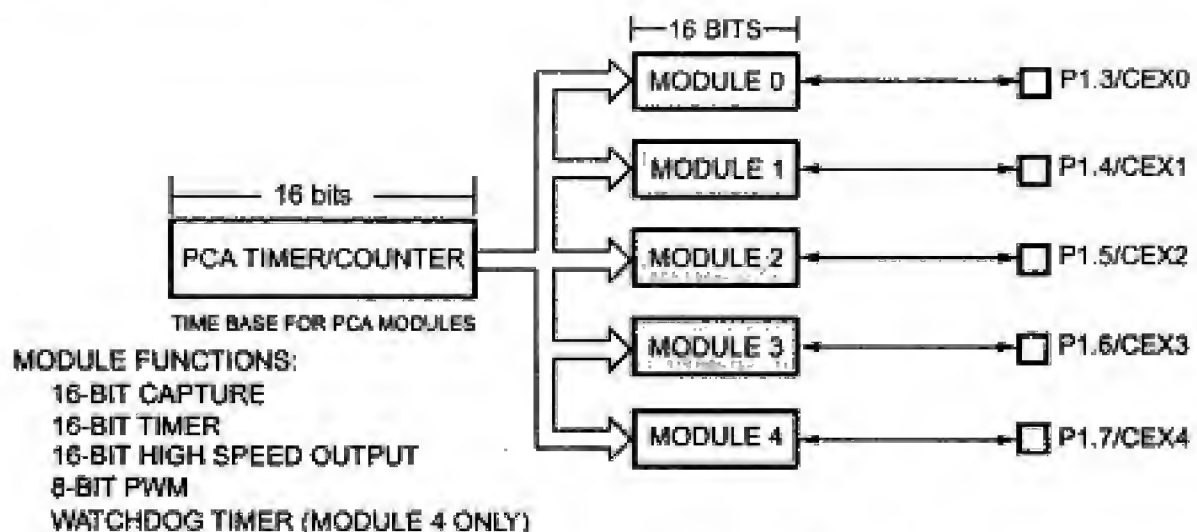


Fig. 16.6 Programmable counter array (PCA)

Each module in the PCA has a special function register associated with it. These registers are : CCAPM0 for module 0, CCAPM1 for module 1, etc. The registers contain the bits that control the mode that each module will operate in.

There are two additional registers associated with each of the PCA modules. They are CCAPnH and CCAPnL and these are the registers that store the 16-bit count when a capture occurs or a compare should occur. When a module is used in the PWM mode these registers are used to control the duty cycle of the output.

PCA Capture Mode

To use one of the PCA modules in the capture mode either one or both of the CCAPM bits CAPN and CAPP for that module must be set. The external CEX input for the module (on port 1) is sampled for a transition. When a valid transition occurs, the PCA hardware loads the value of the PCA counter registers (CH and CL) into the module's capture registers (CCAPnL and CCAPnH). If the CCFn bit for the module in the CCON SFR and the ECCFn bit in the CCAPMn SFR are set then an interrupt will be generated. This is shown in Fig. 16.7.

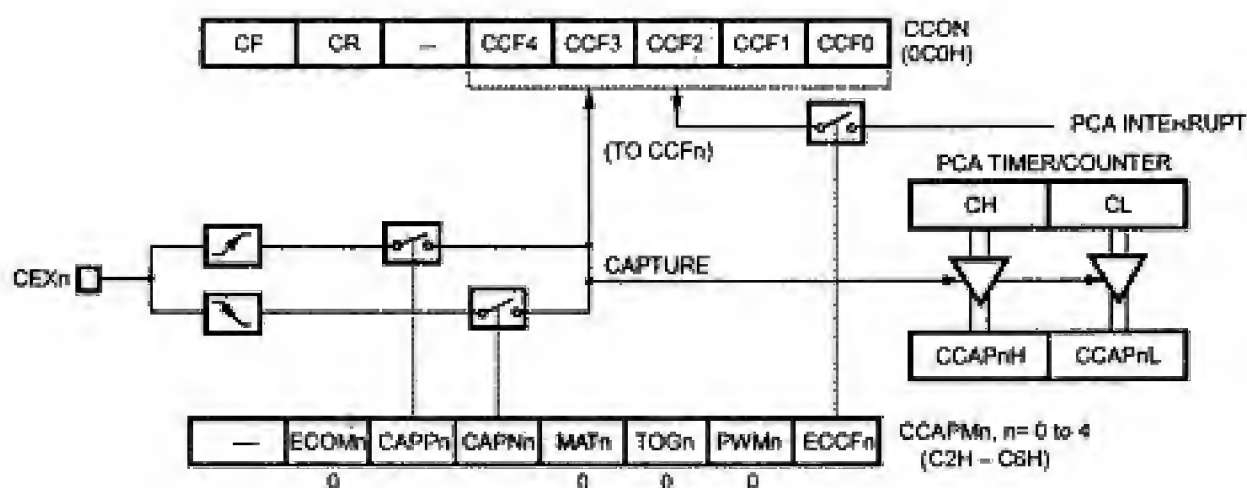


Fig. 16.7 PCA capture mode

16-bit Software Timer Mode

The PCA modules can be used as software timers by setting both the $ECOM$ and MAT bits in the modules $CCAPM_n$ register. The PCA timer will be compared to the module's capture registers and when a match occurs an interrupt will occur if the CCF_n ($CCON$ SFR) and the $ECCF_n$ ($CCAPM_n$ SFR) bits for the module are both set. See Fig. 16.8.

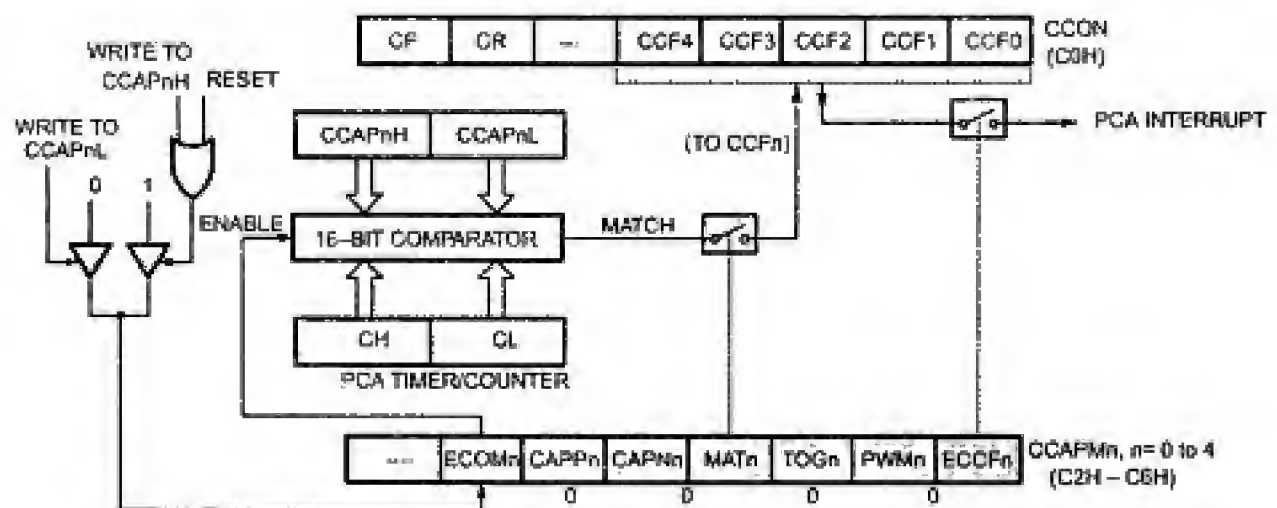


Fig. 16.8 PCA compare mode

High Speed Output Mode

In this mode, the CEX output (on port 1) associated with the PCA module will toggle each time a match occurs between the PCA counter and the module's capture registers. To activate this mode the TOG, MAT, and ECOM bits in the module's CCAPMn SFR must be set. See Fig. 16.9.

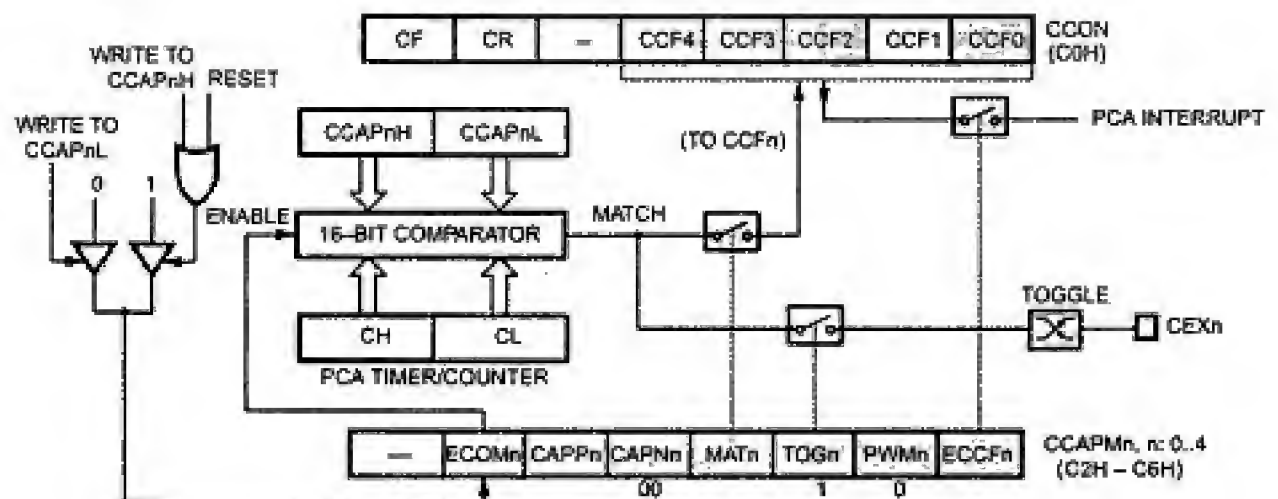


Fig. 16.9 PCA high speed output mode

Pulse Width Modulator Mode

All of the PCA modules can be used as PWM outputs. Fig. 16.10 shows the PWM function. The frequency of the output depends on the source for the PCA timer. All of the modules will have the same frequency of output because they all share the PCA timer.

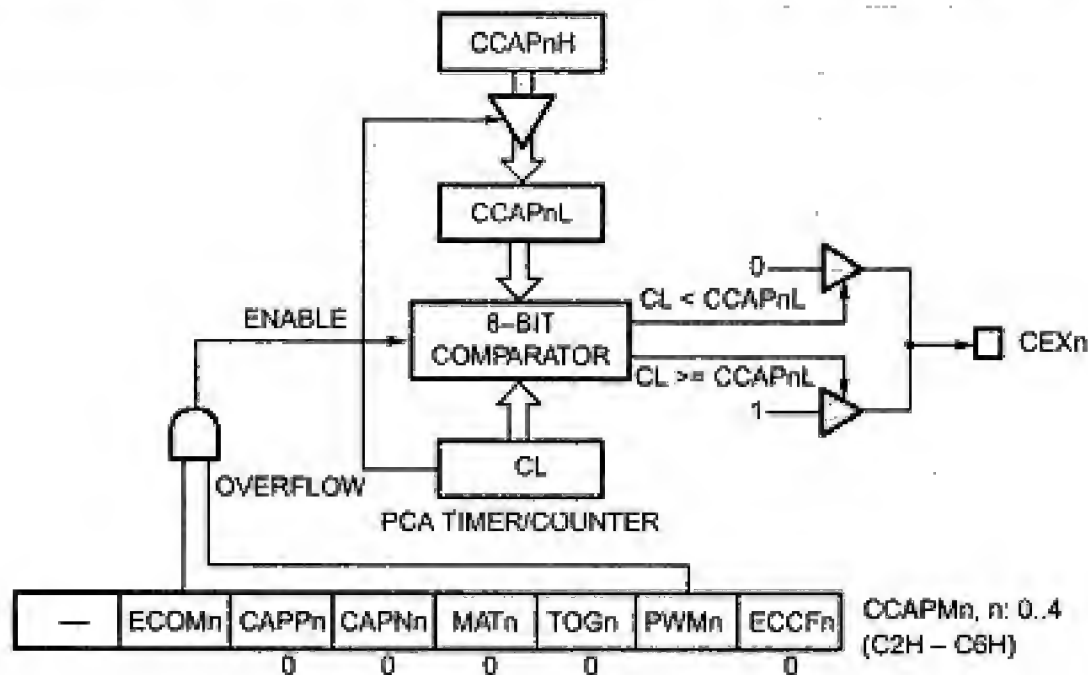


Fig. 16.10 PCA PWM mode

The duty cycle of each module is independently variable using the module's capture register CCAPLn. When the value of the PCA CL SFR is less than the value in the module's CCAPLn SFR the output will be low, when it is equal to or greater than the output will be high. When CL overflows from FF to 00, CCAPLn is reloaded with the value in CCAPnH. This allows updating the PWM without glitches. The PWM and ECOM bits in the module's CCAPMn register must be set to enable the PWM mode.

PCA Watchdog Timer

An on-board watchdog timer is available with the PCA to improve the reliability of the system without increasing chip count. Watchdog timers are useful for systems that are

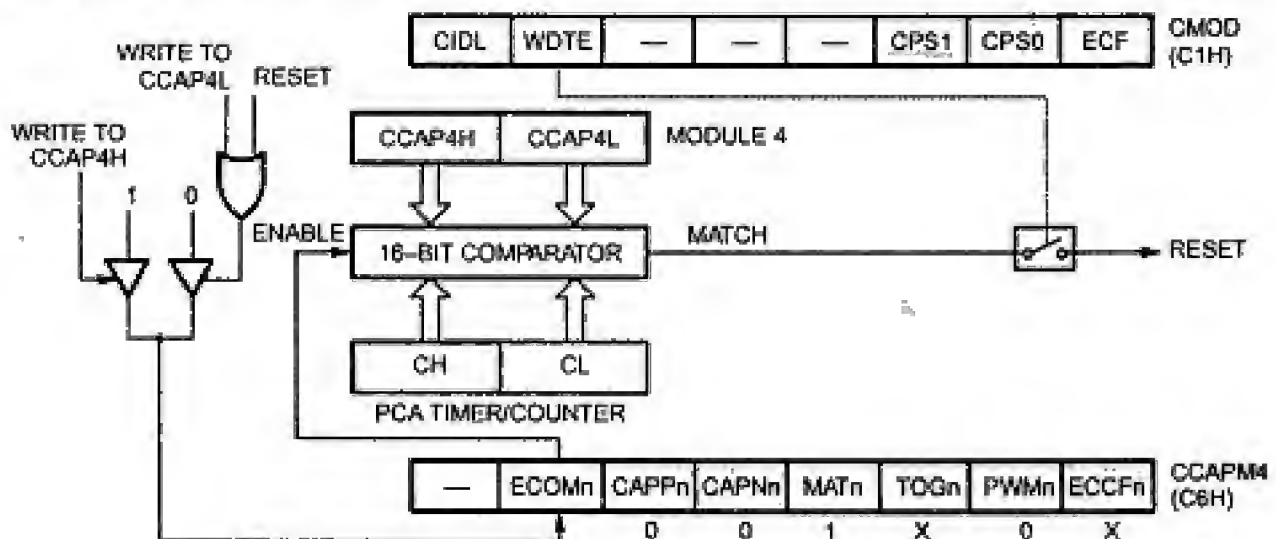


Fig. 16.11 PCA watchdog timer m(module 4 only)

susceptible to noise, power glitches, or electrostatic discharge. Module 4 is the only PCA module that can be programmed as a watchdog. However, this module can still be used for other modes if the watchdog is not needed.

Fig. 16.11 shows a diagram of how the watchdog works. The user pre-loads a 16-bit value in the compare registers. Just like the other compare modes, this 16-bit value is compared to the PCA timer value. If a match is allowed to occur, an internal reset will be generated. This will not cause the RST pin to be driven high.

Expanded Data RAM Addressing

The P89C51RB2/RC2/RD2 has internal data memory that is mapped into four separate segments: the lower 128 bytes of RAM, upper 128 bytes of RAM, 128 bytes Special Function Register (SFR), and 256 bytes expanded RAM (ERAM) (768 bytes for the RD2).

The four segments are :

1. The Lower 128 bytes of RAM (addresses 00H to 7FH) are directly and indirectly addressable.
2. The Upper 128 bytes of RAM (addresses 80H to FFH) are indirectly addressable only.
3. The Special Function Registers, SFRs, (addresses 80H to FFH) are directly addressable only.
4. The 256/768-bytes expanded RAM (ERAM, 00H – 1FFH/2FFH) are indirectly accessed by move external instruction, MOVX, and with the EXTRAM bit cleared.

16.5 Microcontroller Supervisory Control

Microcontroller supervisory control includes :

- Clock circuit
- Reset circuit
- Circuit for demultiplexing of address and data buses
- Buffering of data bus
- Buffering of address bus
- Generating control signals

We have already discussed above circuits in chapter 2 in connection with 8-bit microprocessor. We can use same circuits as supporting circuits for microcontroller 8051, expect circuit for generating control signals. In reset circuits, we have discussed circuit for active high reset and active low reset. 8051 is design for active high reset. Therefore, we have to use active high reset circuitry in 8051 based systems.

8051 provides $\overline{\text{PSEN}}$ signal to access external program memory and provides read ($\overline{\text{RD}}$) and write ($\overline{\text{WR}}$) signals to access external read/write memory. It provides built-in I/O ports and specific address are assigned for them.

Review Questions

1. Write a short note on OTP microcontrollers.
2. List the features of 80C51 microcontroller.
3. What is stop clock mode ?
4. What enhanced UART facility is available in 80C51 UART ?
5. What do you mean by automatic address recognition ?
6. Write short notes on
 - a. Programmable Counter Array (PCA)
 - b. Pulse width modulator
 - c. Watchdog timer
 - d. Expanded data RAM addressing.
7. Write a short note on microcontroller supervisory control.



AVR Microcontrollers

17.1 Introduction

Atmel Corporation is a leading manufacturer of Integrated Circuits (ICs). AVR is the name of a microcontroller series that Atmel produces. Microcontrollers from this series are the RISC (Reduced Instruction Set Computers) microcontrollers. Recall that RISC microprocessors/microcontrollers have very few instructions and instructions are predominantly register-based. The AVR uses Harvard architecture, i.e., they provide separate data and program memory buses. Therefore, AVR controllers have faster execution than 8051.

Features of AVR Family Architecture

1. Provides enhanced RISC architecture with mostly fixed-length instructions, load-store memory access.
2. A RISC instruction set is combined with 32 general purpose working registers. Each register is connected to the ALU, allowing two independent registers to be accessed in a single instruction. This allows additions, subtractions, comparisons to be completed in a single clock cycle.
3. Most of the instructions from the family are executed in only one cycle.
4. Provides two stage instruction pipelining to speed the execution.
5. Provides wide variety of on-chip peripherals such as programmable I/O, ADC, EEPROM, Timer UART, RTC timer and pulse width modulator (PWM).
6. Provides on-chip program and data memory.
7. Provides facility of in-system programmability. In-System Programming (ISP) allows the user to download new code while the microcontroller sits in the application.
8. Available in 8-pin to 64-pin package size to suit wide variety of applications.
9. Provides upto 12 times performance speedup over conventional CISC controllers.
10. Provides wide operation voltage range from 2.7 V - 6.0 V.

In this chapter, we are going to study the architecture and register file of a popular AVR microcontroller AT90S2313.

17.2 Features of AT90S2313 Microcontroller

The AT90S2313 provides the following features :

1. High performance CMOS 8-bit microcontroller based on the AVR enhanced RISC architecture.
2. The device is manufactured using Atmel's high-density nonvolatile memory technology.
3. The AT90S2313 is designed with static logic for operation down to zero frequency.
4. On chip oscillator and clock circuitry.
5. Operating Voltages : 2.7 - 6.0 V (AT90S2313-4) and 4.0 - 6.0 V (AT90S2313-10).
6. Speed Grades : 0 - 4 MHz (AT90S2313-4) and 0 - 10 MHz (AT90S2313-10).
7. The on-chip In-System Programmable Flash allows the program memory to be reprogrammed in-system through an SPI serial interface or by a conventional nonvolatile memory programmer.
8. By combining an enhanced RISC 8-bit CPU with In-System Programmable Flash on a monolithic chip, the Atmel AT90S2313 is a powerful microcontroller that provides a highly flexible and cost-effective solution to many embedded control applications.
9. On-chip program and data memory : 128 bytes EEPROM and 128 bytes SRAM.
10. A rich instruction set combined with 32 general purpose working registers.
11. Each register is connected to the ALU, allowing two independent registers to be accessed in a single instruction. This allows additions, subtractions, comparisons to be completed in a single clock cycle.
12. 15 general-purpose I/O lines.
13. Flexible timer/counters with compare modes.
14. Internal and external interrupts.
15. A programmable serial full duplex UART.
16. 10-bit Pulse Width Modulator (PWM).
17. Programmable Watchdog Timer with internal oscillator.
18. SPI serial port for Flash memory downloading.
19. On-chip Analog Comparator.
20. Provides two software selectable power-saving modes : Idle Mode and Power-down Mode. The Idle Mode stops the CPU while allowing the SRAM, timer/counters, SPI port and interrupt system to continue functioning. The Power-down Mode saves the register contents but freezes the oscillator, disabling all other chip functions until the next external interrupt or hardware reset.

21. The AT90S2313 AVR is supported with a full suite of program and system development tools including: C compilers, macro assemblers, program debugger/simulators, in-circuit emulators and evaluation kits.

17.3 Architecture of AT90S2313

The Fig. 17.1 shows the architecture of AVR AT90S2313 microtroller. As shown in the Fig. 17.1, it consists of instruction register, instruction decoder, program counter, ALU, 32 general purpose working registers (register file), status and test registers, 1 K \times 16 program flash memory, 128 bytes EEPROM and 128 bytes SRAM, 15 general-purpose I/O lines, 8-bit and 16-bit timer/counters with PWM, interrupt unit to handle internal and external interrupts, serial UART, watchdog timer, SPI serial port for Flash memory downloading and analog comparator.

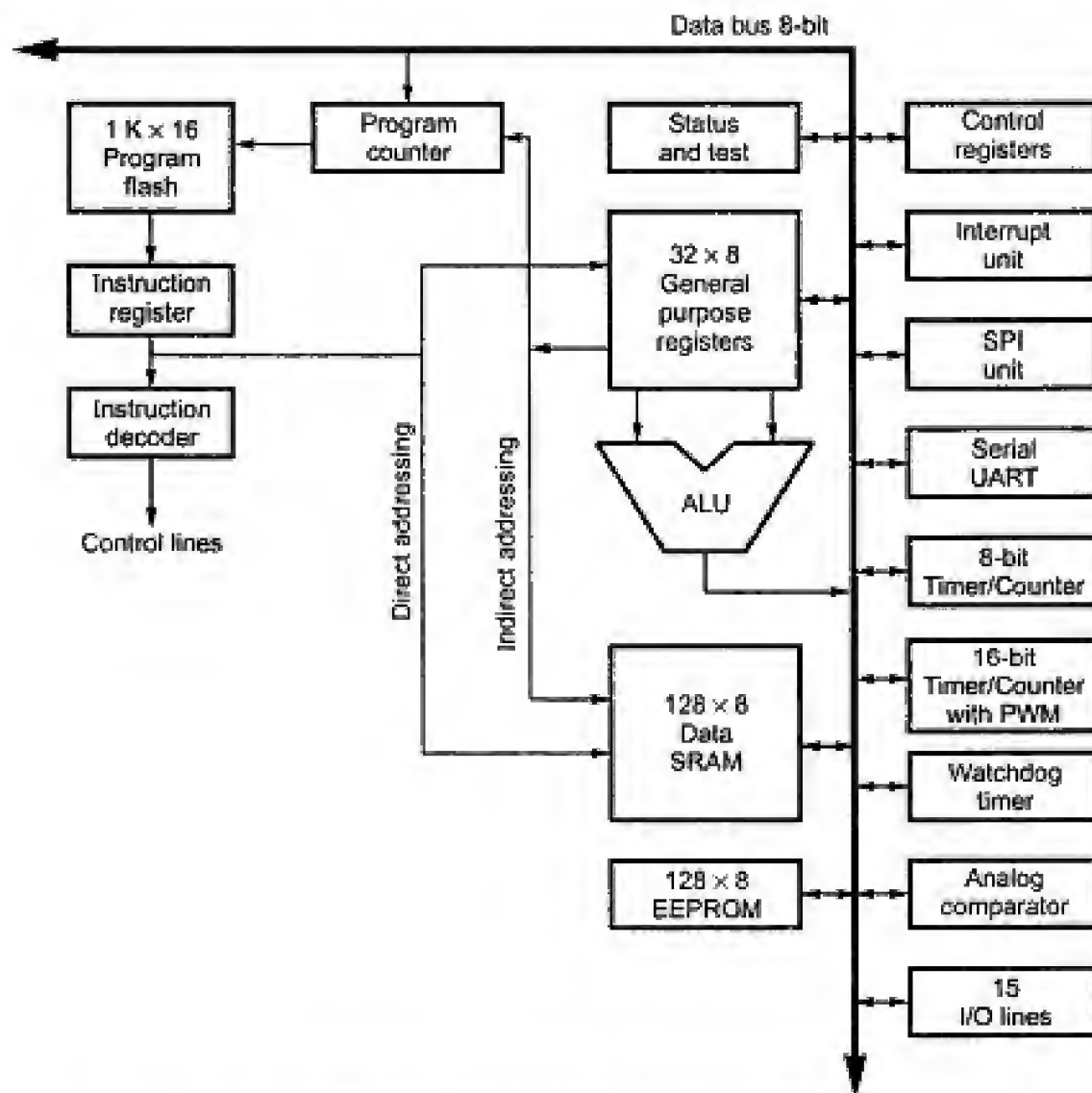


Fig. 17.1 Architecture of AVR AT90S2313 microcontroller

General Purpose Registers (Register File)

It has 32 x 8-bit general-purpose working registers with a single clock cycle access time known as register file. This means that during one single clock cycle, one ALU (Arithmetic Logic Unit) operation is executed. Two operands are output from the register file, the operation is executed and the result is stored back in the register file - in one clock cycle.

Six of the 32 registers can be used as three 16-bit indirect address register pointers for data space addressing. One of the three address pointers is also used as the address pointer for the constant table look-up function. These added function registers are the 16-bit X-register, Y-register and Z-register. In the different addressing modes these address registers have functions as fixed displacement, automatic increment and decrement.

| | 7 0 | | Address | |
|--|--------------------------|--|---------|----------------------|
| | | | | |
| General Purpose Working Registers | R0 | | 00H | |
| | R1 | | 01H | |
| | R2 | | 02H | |
| | ... | | | |
| | R13 | | 0DH | |
| | R14 | | 0EH | |
| | R15 | | 0FH | |
| | R16 | | 10H | |
| | R17 | | 11H | |
| | ... | | | |
| | R26 | | 1AH | X-register low byte |
| | R27 | | 1BH | X-register high byte |
| | R28 | | 1CH | Y-register low byte |
| | R29 | | 1DH | Y-register high byte |
| | R30 | | 1EH | Z-register low byte |
| | R31 | | 1FH | Z-register high byte |

Fig. 17.2 General Purpose Registers

ALU

The ALU supports arithmetic and logic functions between registers or between a constant and a register. Single register operations are also executed in the ALU. The high-performance AVR ALU operates in direct connection with all the 32 general purpose working registers. Within a single clock cycle, ALU operations between registers in the register file are executed. The ALU operations are divided into three main categories - arithmetic, logical and bit functions.

In-System Programmable Flash Program Memory

The AT90S2313 contains 2 kbytes on-chip in-system programmable flash memory for program storage. Since all instructions are 16 or 32-bit words, the Flash is organized as 1 K x 16. The Flash memory has an endurance of at least 1,000 write/erase cycles.

Program Counter

The AT90S2313 Program Counter (PC) is 10 bits wide, thus addressing the 1,024 program memory addresses.

EEPROM Data Memory

The AT90S2313 contains 128 bytes of EEPROM data memory. It is organized as a separate data space in which single bytes can be read and written. The EEPROM has an endurance of at least 100,000 write/erase cycles.

SRAM Data Memory

The Fig. 17.3 shows the organization of data memory of AT90S2313. As shown in the Fig. 17.3, the 224 data memory locations address the Register file, I/O memory and the data SRAM. The first 96 locations address the Register File+ I/O Memory and the next 128 locations address the data SRAM.

Status Register – SREG

The status register contains 8 flag bits that indicate the current state of the microcontroller. At reset all flag bits are at logic 0. These bits can be read or written by the program. The I/O address of status register is 3FH (memory address location is 5FH). The Fig. 17.4 shows the bit positions of status registers.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---------------|-----|-----|-----|-----|-----|-----|-----|-----|------|
| 3FH(5FH) | I | T | H | S | V | N | Z | C | SREG |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

Fig. 17.4 Status register

Various flags of status register (SREG) and their functions are given below :

- Bit 7 – (I) : Global Interrupt Enable :

I = 1 : All interrupts are enabled.

I = 0 : All interrupts are disabled.

Interrupts are enabled according to individual interrupt enable settings in a separate control register only if I-bit = 1. The I-bit is cleared by hardware after an interrupt has occurred and is set by the RETI instruction to enable subsequent interrupts.

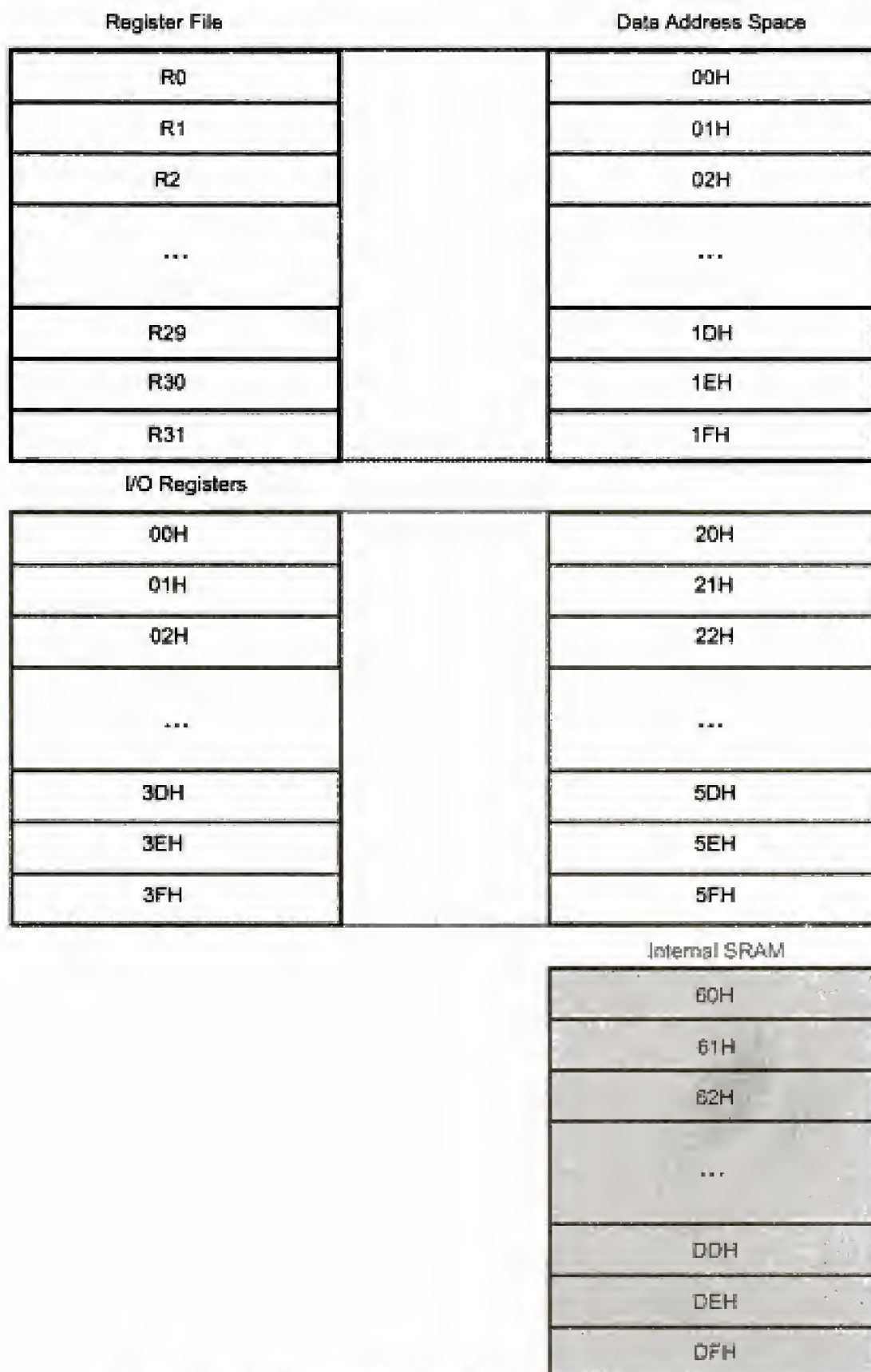


Fig. 17.3 Organization of data memory of AT90S2313

- **Bit 6 – (T) : Bit Copy Storage**

The bit copy instructions BLD (Bit Load) and BST (Bit Store) use the T-bit as source and destination for the operated bit. A bit from a register in the register file can be copied into T by the BST instruction, and a bit in T can be copied into a bit in a register in the register file by the BLD instruction.

- **Bit 5 – (H) : Half-carry Flag**

It indicates a half-carry in some arithmetic operations.

- **Bit 4 – (S) : Sign Bit, $S = N \oplus V$**

The S-bit is always an exclusive or between the negative flag N and the two's complement overflow flag V.

- **Bit 3 – (V) : Two's Complement Overflow Flag**

It supports two's complement arithmetics.

- **Bit 2 – (N) : Negative Flag**

It indicates a negative result after the different arithmetic and logic operations.

- **Bit 1 – (Z) : Zero Flag**

It indicates a zero result after the different arithmetic and logic operations.

- **Bit 0 – (C) : Carry Flag**

17.4 Memory Map of AVR AT90S2313

The Fig. 17.5 shows the internal memory map of AVR AT90S2313. It consists of program flash memory of (1 K × 16), and data memory (32 general purpose working registers + 64 I/O registers + 128 × 8 SRAM + 128 × 8 EEPROM).

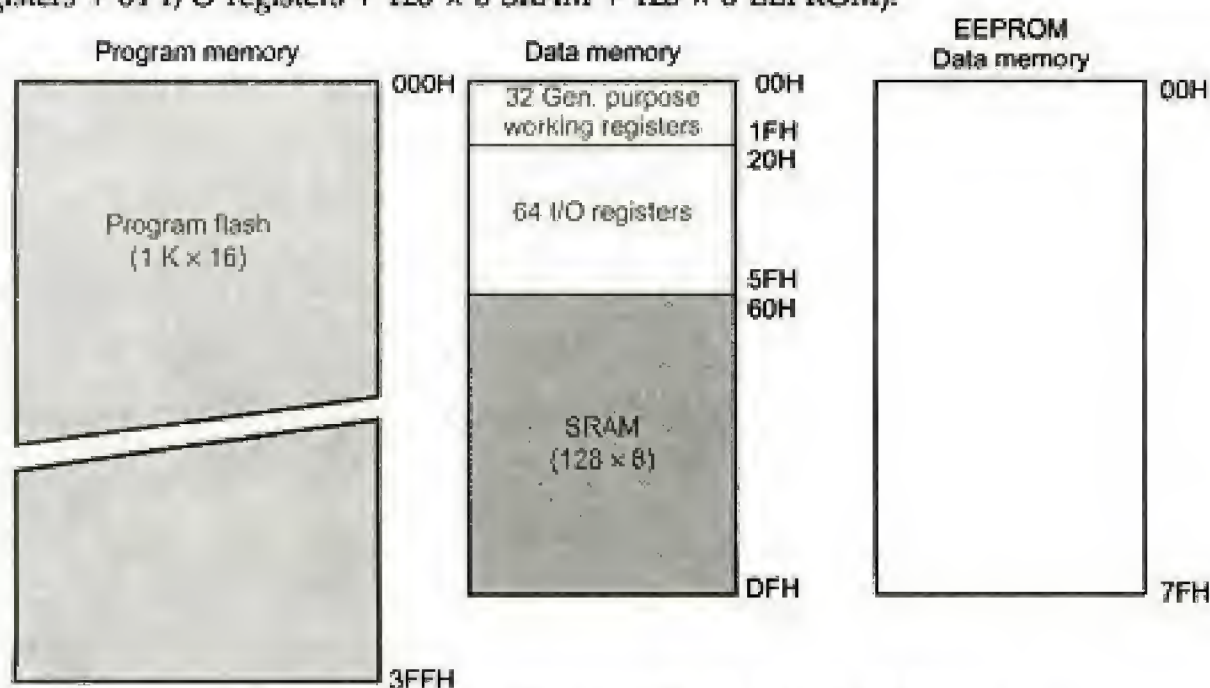


Fig. 17.5 Internal memory map of AVR AT90S2313

17.5 Memory Access and Instruction Execution

The AVR AT90S2313 uses two stage pipelining. The Fig. 17.6 shows the parallel instruction fetches and instruction executions enabled by the Harvard architecture and the fast-access register file concept. This is the basic pipelining concept to obtain up to 1 MIPS.

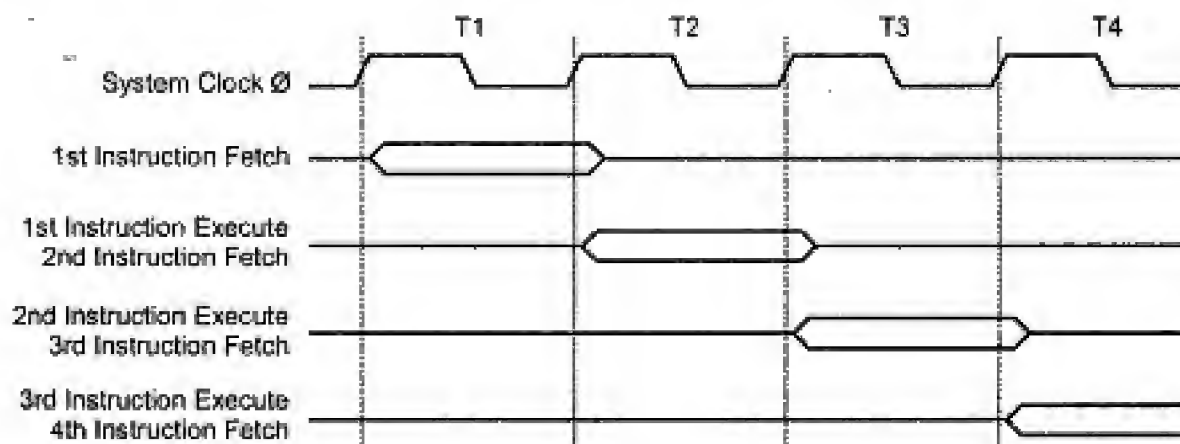


Fig. 17.6 The parallel instruction fetches and instruction executions

The Fig. 17.7 shows the internal timing concept for the register file. In a single clock cycle an ALU operation using two register operands is executed and the result is stored back to the destination register.

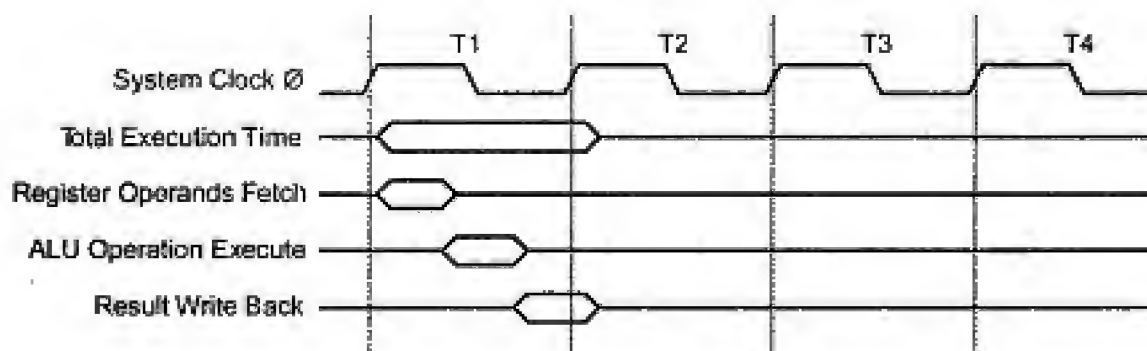


Fig. 17.7 Single-cycle ALU Operation

The internal data SRAM access is performed in two system clock cycles. This is illustrated in Fig. 17.8.

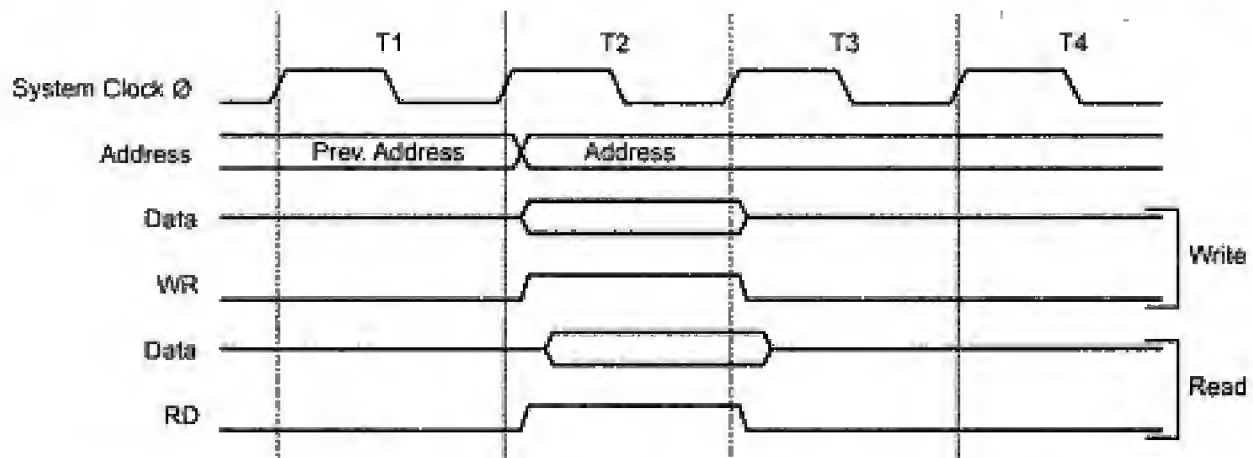


Fig. 17.8 Internal data SRAM access

17.6 I/O Memory

All AT90S2313 I/O and peripherals are placed in the I/O space. The I/O locations are accessed by the IN and OUT instructions transferring data between the 32 general-purpose working registers and the I/O space. I/O registers within the address range 00H - 1FH are directly bit-accessible using the SBI and CBI instructions. In these registers, the value of single bits can be checked by using the SBIS and SBIC instructions. The I/O memory can be accessed in two ways - as an I/O register using the I/O specific commands or as a SRAM. When using the I/O specific commands IN and OUT, the I/O addresses 00H - 3FH must be used. When addressing I/O registers as SRAM, 20H must be added to this address. The Table 17.1 shows the I/O space definition of the AT90S2313.

| Address Hex | Name | Function |
|-------------|--------|---------------------------------------|
| 3FH (5FH) | SREG | Status Register |
| 3DH (5DH) | SPL | Stack Pointer Low |
| 3BH (5BH) | GI_MSK | General Interrupt Mask Register |
| 3AH (5AH) | GIFR | General Interrupt Flag Register |
| 39H (59H) | TIMSK | Timer/Counter Interrupt Mask Register |
| 38H (58H) | TIFR | Timer/Counter Interrupt Flag Register |
| 35H (55H) | MCUCR | MCU general Control Register |
| 33H (53H) | TCCR0 | Timer/Counter 0 Control Register |
| 32H (52H) | TCNT0 | Timer/Counter 0 (8-bit) |
| 2FH (4FH) | TCCR1A | Timer/Counter 1 Control Register A |
| 2EH (4EH) | TCCR1B | Timer/Counter 1 Control Register B |

| | | |
|-----------|--------|---|
| 2DH (4DH) | TCNT1H | Timer/CouNter 1 high Byte |
| 2CH (4CH) | TCNT1L | Timer/CouNter 1 Low Byte |
| 2BH (4BH) | OCR1AH | Output Compare Register 1 High Byte |
| 2AH (4AH) | OCR1AL | Output Compare Register 1 Low Byte |
| 25H (45H) | ICR1H | T/C 1 Input Capture Register High Byte |
| 24H (44H) | ICR1L | T/C 1 Input Capture Register Low Byte |
| 21H (41H) | WDTCSR | WatchDog Timer Control Register |
| 1EH (3EH) | EEAR | EEPROM Address Register |
| 1DH (3DH) | EEDR | EEPROM Data Register |
| 1CH (3CH) | EECR | EEPROM Control Register |
| 18H (38H) | PORTB | Data Register, Port B |
| 17H (37H) | DDRB | Data Direction Register, Port B |
| 16H (36H) | PINB | Input Pins, Port B |
| 12H (32H) | PORTD | Data Register, Port D |
| 11H (31H) | DDRD | Data Direction Register, Port D |
| 10H (30H) | PIND | Inputs Pins, Port D |
| 0CH(2CH) | UDR | UART I/O Data Register |
| 0BH (2BH) | USR | UART Status Register |
| 0AH (2AH) | UCR | UART Control Register |
| 09H (29H) | UBRR | UART Baud Rate Register |
| 08H (28H) | ACSR | Analog Comparator Control and Status Register |

Table 17.1 I/O space definition of the AT90S2313

Note : Reserved and unused locations are not shown in the table.

17.7 Reset and Interrupt Handling

The AT90S2313 provides 10 different interrupt sources. These interrupts and the separate reset vector each have a separate program vector in the program memory space. All the interrupts are assigned individual enable bits that must be set (one) together with the I - bit in the Status Register in order to enable the interrupt. The lowest addresses in the program memory space are automatically defined as the Reset and Interrupt vectors. The Table 17.2 shows the complete list of vectors allotted to interrupts. The list also determines the priority levels of the different interrupts. The lower the address, the higher

the priority level. RESET has the highest priority, and next is INT0 (the External Interrupt Request 0), etc.

| Vector No. | Program Address | Source | Interrupt Definition |
|------------|-----------------|--------------|---|
| 1 | 000H | RESET | Hardware Pin, Power-on Reset and Watchdog Reset |
| 2 | 001H | INT0 | External Interrupt Request 0 |
| 3 | 002H | INT1 | External Interrupt Request 1 |
| 4 | 003H | TIMER1 CAPT1 | Timer/Counter 1 Capture Event |
| 5 | 004H | TIMER1 COMP1 | Timer/Counter1 Compare Match |
| 6 | 005H | TIMER1 OVF1 | Timer/Counter1 Overflow |
| 7 | 006H | TIMER0 OVF0 | Timer/Counter0 Overflow |
| 8 | 007H | UART, RX | UART, RX Complete |
| 9 | 008H | UART, UDRE | UART Data Register Empty |
| 10 | 009H | UART, TX | UART, TX Complete |
| 11 | 00AH | ANA_COMP | Analog Comparator |

Table 17.2 Reset and interrupt vectors

The AT90S2313 has three sources of reset :

- Power-on Reset
- External Reset
- Watchdog Reset

17.8 Registers in AT90S2313

17.8.1 General Interrupt Mask Register - GIMSK

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---------------|------|------|---|---|---|---|---|---|
| 3BH(5BH) | INT1 | INT0 | – | – | – | – | – | – |
| Read/Write | R/W | R/W | R | R | R | R | R | R |
| Initial value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Fig. 17.9

- Bit 7 – INT1 : External Interrupt Request 1 Enable.

When the INT1 bit is set (one) and the I-bit in the Status Register (SREG) is set (one), the external pin interrupt is enabled.

- Bit 6 – INT0 : External Interrupt Request 0 Enable.

When the INT0 bit is set (one) and the I-bit in the Status Register (SREG) is set (one), the external pin interrupt is enabled.

- Bits 5 - 0 : Reserved Bits.

17.8.2 General Interrupt FLAG Register – GIFR

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---------------|-------|-------|---|---|---|---|---|---|
| 3AH(5AH) | INTF1 | INTF0 | – | – | – | – | – | – |
| Read/Write | R/W | R/W | R | R | R | R | R | R |
| Initial value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Fig. 17.10

- Bit 7/6 – INTF1/0 : External Interrupt Flag 1/0.

When an edge on the INT1/0 pin triggers an interrupt request, the corresponding interrupt flag, INTF1/0, becomes set (one). If the I-bit in SREG and the corresponding interrupt enable bit, INT1/0 bit in GIMSK, are set (one), the MCU will jump to the interrupt vector. The flag is cleared when the interrupt routine is executed. Alternatively, the flag can be cleared by writing a logical “1” to it.

- Bits 5 - 0 : Reserved Bits.

17.8.3 Timer/Counter Interrupt Mask Register – TIMSK

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---------------|-------|--------|---|---|--------|---|-------|---|
| 39H(59H) | TOIE1 | OCIE1A | – | – | TICIE1 | – | TOIE0 | – |
| Read/Write | R/W | R/W | R | R | R/W | R | R/W | R |
| Initial value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Fig. 17.11

- Bit 7 – TOIE1: Timer/Counter1 Overflow Interrupt Enable.

When the TOIE1 bit is set (one) and the I-bit in the Status Register is set (one), the Timer/Counter1 Overflow Interrupt is enabled.

- Bit 6 – OCIE1A: Timer/Counter1 Output Compare Match Interrupt Enable.

When the OCIE1A bit is set (one) and the I-bit in the Status Register is set (one), the Timer/Counter1 Compare Match Interrupt is enabled.

- Bit 5 - 4 : Reserved Bits.
- Bit 3 – TICIE1 : Timer/Counter1 Input Capture Interrupt Enable.

When the TICIE1 bit is set (one) and the I-bit in the Status Register is set (one), the Timer/Counter1 Input Capture Event Interrupt is enabled.

- Bit 2 : Reserved Bit.
- Bit 1 – TOIE0 : Timer/Counter0 Overflow Interrupt Enable.

When the TOIE0 bit is set (one) and the I-bit in the Status Register is set (one), the Timer/Counter0 Overflow Interrupt is enabled.

- Bit 0 : Reserved Bit.

17.8.4 Timer/Counter Interrupt FLAG Register – TIFR

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---------------|------|-------|---|---|------|---|------|---|
| 38H(58H) | TOV1 | OCF1A | – | – | ICF1 | – | TOV0 | – |
| Read/Write | R/W | R/W | R | R | R/W | R | R/W | R |
| Initial value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Fig. 17.12

- Bit 7 – TOV1: Timer/Counter1 Overflow Flag.

The TOV1 is set (one) when an overflow occurs in Timer/Counter1.

- Bit 6 – OCF1A : Output Compare Flag 1A.

The OCF1A bit is set (one) when a compare match occurs between the Timer/Counter1 and the data in OC1A (Output Compare Register 1A).

- Bits 5 - 4 : Reserved Bits.
- Bit 3 – ICF1: Input Capture Flag 1.

The ICF1 bit is set (one) to flag an input capture event, indicating that the Timer/Counter1 value has been transferred to the Input Capture Register (ICR1).

- Bit 2 : Reserved Bit.
- Bit 1 – TOV0 : Timer/Counter0 Overflow Flag

The bit TOV0 is set (one) when an overflow occurs in Timer/Counter0.

- Bit 0 : Reserved Bit.

Note : Flags are cleared by hardware when executing the corresponding interrupt handling vector. Alternatively, they are cleared by writing a logical “1” to the flag.

17.8.5 MCU Control Register – MCUCR

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---------------|---|---|-----|-----|-------|-------|-------|-------|
| 35H(55H) | – | – | SE | SM | ISC11 | ISC10 | ISC01 | ISC00 |
| Read/Write | R | R | R/W | R/W | R/W | R/W | R/W | R/W |
| Initial value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Fig. 17.13

The MCU Control Register contains control bits for general MCU functions.

- Bits 7 - 6 : Reserved Bits.
- Bit 5 – SE : Sleep Enable.

The SE bit must be set (one) to make the MCU enter the Sleep Mode when the SLEEP instruction is executed.

- Bit 4 – SM : Sleep Mode

This bit selects between the two available sleep modes.

SM = 0 Idle Mode.

SM = 1 Power-down Mode.

- Bits 3, 2 – ISC11, ISC10 : Interrupt Sense Control 1 Bit 1 and Bit 0

| ISC11 | ISC10 | Description |
|-------|-------|---|
| 0 | 0 | The low level of INT1 generates an interrupt request |
| 0 | 1 | Reserved |
| 1 | 0 | The falling edge of INT1 generates an interrupt request |
| 1 | 1 | The rising edge of INT1 generates an interrupt request |

- Bits 1, 0 – ISC01, ISC00 : Interrupt Sense Control 0 Bit 1 and Bit 0

| ISC01 | ISC00 | Description |
|-------|-------|---|
| 0 | 0 | The low level of INT0 generates interrupt request |
| 0 | 1 | Reserved |
| 1 | 0 | The falling edge of INT0 generates an interrupt request |
| 1 | 1 | The rising edge of INT0 generates an interrupt request |

17.8.6 Timer/Counters Control Registers

The AT90S2313 provides two general-purpose Timers/Counters - one 8-bit T/C and one 16-bit T/C. The Timer/Counters have individual prescaling selection from the same

10-bit prescaling timer. Both Timer/Counters can either be used as a timer with an internal clock time base or as a counter with an external pin connection that triggers the counting. Let us see the control registers for timer/counters.

17.8.6.1 Timer/Counter1 Control Register A – TCCR1A

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---------------|--------|--------|---|---|---|---|-------|-------|--------|
| 2FH(4FH) | COM1A1 | COM1A0 | – | – | – | – | PWM11 | PWM10 | TCCR1A |
| Read/Write | R/W | R/W | R | R | R | R | R/W | R/W | |
| Initial value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

Fig. 17.14

- Bits 7,6 – COM1A1, COM1A0: Compare Output Mode1, Bits 1 and 0

The COM1A1 and COM1A0 control bits determine any output pin action following a compare match in Timer/Counter1. Any output pin actions affect pin OC1 (Output Compare pin 1) (PB3). This is an alternative function to the I/O port, and the corresponding direction control bit must be set (one) to control an output pin. The control configuration is shown in Table 17.3.

| COM1A1 | COM1A0 | Description |
|--------|--------|---|
| 0 | 0 | Timer/Counter1 disconnected from output pin OC1 |
| 0 | 1 | Toggle the OC1 output line. |
| 1 | 0 | Clear the OC1 output line (to zero). |
| 1 | 1 | Set the OC1 output line (to one). |

Table 17.3 Compare 1 mode select

- Bits 5-2 – Reserved Bits
- Bits 1,0 – PWM11, PWM10: Pulse Width Modulator Select Bits

These bits select PWM operation of Timer/Counter1 as specified in Table 17.4.

| PWM11 | PWM10 | Description |
|-------|-------|---|
| 0 | 0 | PWM operation of Timer/Counter1 is disabled |
| 0 | 1 | Timer/Counter1 is an 8-bit PWM |
| 1 | 0 | Timer/Counter1 is a 9-bit PWM |
| 1 | 1 | Timer/Counter1 is a 10-bit PWM |

Table 17.4 PWM mode select

17.8.6.2 Timer/Counter1 Control Register B – TCCR1B

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---------------|-------|-------|---|---|------|------|------|------|--------|
| 2EH(4EH) | ICNC1 | ICES1 | – | – | CTC1 | CS12 | CS11 | CS10 | TCCR1B |
| Read/Write | R/W | R/W | R | R | R/W | R/W | R/W | R/W | |
| Initial value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

Fig. 17.15

- **Bit 7 – ICNC1 : Input Capture1 Noise Canceller (4 CKs)**

When the ICNC1 bit is cleared (zero), the input capture trigger noise canceller function is disabled.

- **Bit 6 – ICES1 : Input Capture1 Edge Select**

While the ICES1 bit is cleared (zero), the Timer/Counter1 contents are transferred to the Input Capture Register (ICR1) on the falling edge of the input capture pin (ICP). While the ICES1 bit is set (one), the Timer/Counter1 contents are transferred to the Input Capture Register (ICR1) on the rising edge of the input capture pin (ICP).

- **Bits 5, 4 : Reserved Bits**

- **Bit 3 – CTC1 : Clear Timer/Counter1 on Compare Match**

When the CTC1 control bit is set (one), the Timer/Counter1 is reset to 0000H in the clock cycle after a compareA match. If the CTC1 control bit is cleared, Timer/Counter1 continues counting and is unaffected by a compare match.

- **Bits 2,1,0 – CS12, CS11, CS10 : Clock Select1, Bits 2, 1 and 0**

The Clock Select1 bits 2, 1 and 0 define the prescaling source of Timer/Counter1, as shown in Table 17.5.

| CS12 | CS11 | CS10 | Description |
|------|------|------|--------------------------------------|
| 0 | 0 | 0 | Stop, the Timer/Counter1 is stopped. |
| 0 | 0 | 1 | CK |
| 0 | 1 | 0 | CK/8 |
| 0 | 1 | 1 | CK/64 |
| 1 | 0 | 0 | CK/256 |
| 1 | 0 | 1 | CK/1024 |
| 1 | 1 | 0 | External Pin T1, falling edge |
| 1 | 1 | 1 | External Pin T1, rising edge |

Table 17.5 Clock 1 prescale select

17.8.6.3 Timer/Counter1 – TCNT1H and TCNT1L

| | | | | | | | | | |
|---------------|-----|-----|-----|-----|-----|-----|-----|-----|------------|
| Bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | |
| 20H(4DH) | MSB | | | | | | | | TCNT1 H |
| 2CH(4CH) | | | | | | | | LSB | TCNT1L |
| | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

Fig. 17.16

This 16-bit register contains the prescaled value of the 16-bit Timer/Counter1.

17.8.6.4 Timer/Counter1 Output Compare Register A – OCR1AH and OCR1AL

| | | | | | | | | | |
|---------------|-----|-----|-----|-----|-----|-----|-----|-----|------------|
| Bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | |
| 2BH(4BH) | MSB | | | | | | | | OCR1A H |
| 2AH(4AH) | | | | | | | | LSB | OCR1AL |
| | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

Fig. 17.17

The output compare register is a 16-bit read/write register. The Timer/Counter1 Output Compare Register contains the data to be continuously compared with Timer/Counter1.

17.8.6.5 Timer/Counter1 Input Capture Register – ICR1H and ICR1L

| Bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | |
|---------------|-----|----|----|----|----|----|---|-----|-------|
| 25H(45H) | MSB | | | | | | | | ICR1H |
| 24H(44H) | | | | | | | | LSB | ICR1L |
| | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| Read/Write | R | R | R | R | R | R | R | R | |
| | R | R | R | R | R | R | R | R | |
| Initial value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

Fig. 17.18

The input capture register is a 16-bit read-only register. When the rising or falling edge (according to the input capture edge setting [ICES1]) of the signal at the input capture pin (ICP) is detected, the current value of the Timer/Counter1 is transferred to the Input Capture Register (ICR1). At the same time, the input capture flag (ICF1) is set (one).

17.8.7 Watchdog Timer Control Register – WDTCSR

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---------------|---|---|---|-------|-----|------|------|------|--------|
| 21H (41H) | – | – | – | WDTOE | WDE | WDP2 | WDP1 | WDP0 | WDTCSR |
| Read/Write | R | R | R | R/W | R/W | R/W | R/W | R/W | |
| Initial value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

Fig. 17.19

- Bits 7-5 – Reserved Bits
- Bit 4 – WDTOE : Watchdog Turn-off Enable

This bit must be set (one) when the WDE bit is cleared. Otherwise, the Watchdog will not be disabled. Once set, hardware will clear this bit to zero after four clock cycles. Refer to the description of the WDE bit for a Watchdog disable procedure.

- Bit 3 – WDE : Watchdog Enable

When the WDE is set (one) the Watchdog Timer is enabled, and if the WDE is cleared (zero), the Watchdog Timer function is disabled.

- Bits 2-0 – WDP2, WDP1, WDP0 : Watchdog Timer Prescaler 2, 1 and 0

The WDP2, WDP1 and WDP0 bits determine the Watchdog Timer prescaling when the Watchdog Timer is enabled. The different prescaling values and their corresponding time-out periods are shown in Table 17.6.

| WDP2 | WDP1 | WDP0 | Number of WDT Oscillator Cycles | Typical Time-out at $V_{CC} = 3.0\text{ V}$ | Typical Time-out at $V_{CC} = 5.0\text{ V}$ |
|------|------|------|---------------------------------|---|---|
| 0 | 0 | 0 | 16K cycles | 47 ms | 15 ms |
| 0 | 0 | 1 | 32K cycles | 94 ms | 30 ms |
| 0 | 1 | 0 | 64K cycles | 0.19 s | 60 ms |
| 0 | 1 | 1 | 128K cycles | 0.38 s | 0.12 s |
| 1 | 0 | 0 | 256K cycles | 0.75 s | 0.24 s |
| 1 | 0 | 1 | 512K cycles | 1.5 s | 0.49 s |
| 1 | 1 | 0 | 1,024K cycles | 3.0 s | 0.97 s |
| 1 | 1 | 1 | 2,048K cycles | 6.0 s | 1.9 s |

Table 17.6 Watchdog Timer Prescale Select

17.8.8 EEPROM Read/Write Access Registers

The EEPROM access registers are accessible in the I/O space.

17.8.8.1 EEPROM Address Register – EEAR

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---------------|---|-------|-------|-------|-------|-------|-------|-------|------|
| 1EH (3EH) | – | EEAR6 | EEAR5 | EEAR4 | EEAR3 | EEAR2 | EEAR1 | EEAR0 | EEAR |
| Read/Write | R | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

Fig. 17.20

- Bit 7-Reserved Bit
- Bit 6-0 – EEAR6-0 : EEPROM Address

The EEPROM Address Register (EEAR6-0) specifies the EEPROM address in the 128 bytes EEPROM space.

17.8.8.2 EEPROM Data Register – EEDR

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---------------|-----|-----|-----|-----|-----|-----|-----|-----|------|
| 1DH (3DH) | MSB | | | | | | | LSB | EEDR |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

Fig. 17.21

- Bit 7-0 – EEDR7-0 : EEPROM Data

17.8.8.3 EEPROM Control Register – EECR

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---------------|---|---|---|---|---|-------|------|------|------|
| 1CH (3CH) | – | – | – | – | – | EEMWE | EEWE | EERE | EECR |
| Read/Write | R | R | R | R | R | R/W | R/W | R/W | |
| Initial value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

Fig. 17.22

- Bit 7-3-Reserved Bits

Review Questions

1. State the features of AVR family architecture.
2. State the features of AT90S2313 microcontroller.
3. Draw and explain the architecture of AT90S2313 microcontroller.
4. Draw and explain the organization of data memory of AT90S2313 microcontroller.
5. Draw and explain the internal memory map of AT90S2313 microcontroller.
6. Explain various flags of status register of AT90S2313 microcontroller.
7. Explain memory access and instruction execution related to AT90S2313 microcontroller.
8. Write a short note on I/O memory of AT90S2313 microcontroller.
9. Explain various sources of interrupts in AT90S2313 microcontroller.
10. List the sources of resets in AT90S2313 microcontroller.
11. Explain any two register formats of AT90S2313 microcontroller.



18

Introduction to PIC Microcontroller

18.1 Introduction

In chapter 7, we have studied a microcontroller 8051. Peripheral Interface Controller (PIC) is a term introduced by Microchip Technology. It is a family of low-cost, high performance, CMOS, fully-static microcontrollers. They have interesting and attractive features which are suitable for a wide range of applications. A number of PIC parts and versions are available in the market. In this chapter, first we will study the newly introduced features of PIC microcontroller. Then, our focus will be the PIC16C6X and PIC16C7X family.

18.2 Features of PIC Microcontrollers

The features of PIC microcontrollers because of which they have been successful in the market-place are discussed here.

1. Harvard Architecture

PIC uses Harvard architecture and they are high performance RISC processors. Harvard architecture has the program memory and data memory as separate memories and are accessed from separate buses. This improves bandwidth over traditional Von-Neumann architecture in which program and data are fetched from the same memory using the same bus. To execute an instruction, a Von-Neumann machine must make one or more (generally more) accesses across the 8-bit bus to fetch the instruction. Then data may need to be fetched, operated on, and possibly written. As can be seen from this description, that bus can be extremely congested. While with a Harvard architecture, the instruction is fetched in a single instruction cycle (all 14-bits). While the program memory is being accessed, the data memory is on an independent bus and can be read and written. These separated buses allow one instruction to execute while the next instruction is fetched. A comparison of Harvard vs. Von-Neumann architectures is shown in Fig. 18.1.

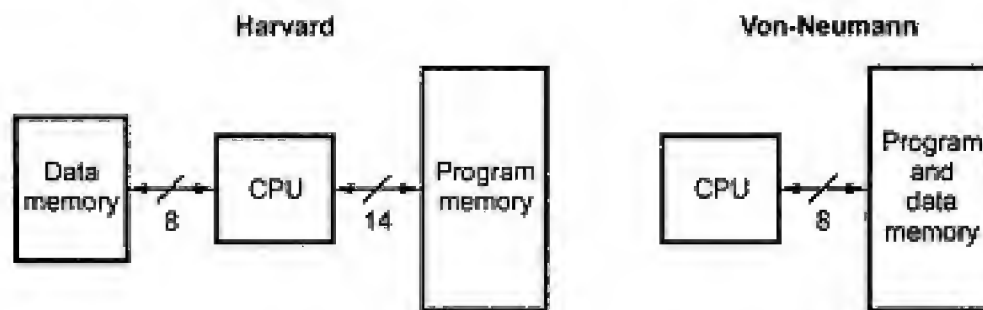


Fig. 18.1 Harvard Vs. Von-Neumann block architectures

The 8051 uses Von-Neumann architecture. On the other hand, AVR and PIC microcontrollers use Harvard architecture.

2. Register File Architecture

The register files/data memory can be directly or indirectly addressed. All special function registers, including the program counter, are mapped in the data memory.

3. Instruction Set Simplicity

a. Reduced instruction set :

The instruction set of PIC consists of only 35 instructions.

b. Orthogonal (symmetric) instructions :

The instructions are orthogonal. It is possible to carry out any operation on any register using any addressing mode when the instructions are orthogonal. The format of almost all instructions is same. So all PIC registers and addressing modes can be used interchangeably. Also, the choices of opcode, register and addressing mode are mutually independent. The instruction set is well designed and highly orthogonal, so the required tasks can be performed with fewer instructions.

The programmer can easily learn and remember such an instruction set with fewer instructions and programming can be done quickly and efficiently.

c. Single cycle instructions :

The program memory bus is 14-bits wide in PIC. The entire instruction is fetched in a single machine cycle. All the required information is contained within an instruction and is executed in a single cycle.

d. High speed of instruction execution :

When PIC is operated at its maximum clock rate, almost each instruction can be executed within 0.2 μ s or five instructions can be executed per microsecond.

e. Less number of pulses per machine cycle :

We have seen that machine cycle in microcontroller 8051 consists of 12 clock pulses. The machine cycle of PIC consists of only 4 clock pulses.

f. Single word instructions :

Typically in Von-Neumann architecture, most instructions are multibyte. Each instruction may take multiple bytes, so there is no assurance that each location is a valid instruction.

PIC uses Harvard architecture and single word instructions. The opcodes are 14-bits wide. A 14-bit wide program memory access bus fetches a 14-bit instruction in a single cycle. The number of words of program memory locations equals the number of instructions for the device. This means that all locations are valid instructions.

g. Long word instructions :

PIC instruction set contains long word instructions. Long word instructions have a wider (more bits) instruction bus than 8-bit Data Memory Bus. These two different width buses are possible because of Harvard architecture. Also, it allows instructions to be sized differently than 8-bit wide data word. The program memory width is optimized to the architectural requirements and hence the program memory can be used more efficiently.

4. EPROM Selectable Operating Frequency

A low cost RC circuit or a quartz crystal or a ceramic resonator is used for the generation of clock. In PIC, the oscillator options are EPROM selectable. The fully static design of PIC allows the selection of operating frequency from the specified range of frequencies. The oscillator clock may be stopped at any instant and may be restored back.

5. Integration of Operational Features**a. Built-in power-on-reset**

PIC has a built-in power-on-reset.

b. Brown-out-reset

PIC has a brown-out-reset. A brown-out-reset feature causes a reset of the PIC when the power supply voltage drops below 4 V or so.

Power-on-reset and brown-out-reset protections ensure that the PIC-chip operates only when the supply voltage is within specification.

c. Watchdog timer (WDT)

Watchdog timer is a special timer with specific function. It is used to prevent software crashes, i.e. endless loop.

d. Power saving SLEEP mode

The PIC can put itself to sleep to save power during intervals when it has nothing to do. Thus, PIC supports a power saving SLEEP mode. A software command allows the PIC to enter into this mode. The PIC remains in SLEEP mode till it is reset again.

6. Programmable Timer Options

There are three versatile timers in PIC which perform the following functions.

- Characterize inputs
- Control outputs
- Provide internal timing for program execution

7. Interrupt Control

The PIC can control up to 12 independent interrupt sources.

8. ADC Function

Analog to digital conversion function is supported by most of the PICs.

9. Powerful Output Pin Control

A single instruction of PIC can select and drive a single output pin high or low in its instruction execution time (0.2 μ s). A load of up to 25 mA can be driven by this pin.

10. I/O Port Expansion

There is a built-in serial peripheral interface.

11. Serial Programming via Two Pins**12. EPROM/OTP/ROM Options**

Development in PIC is supported by Erasable Programmable Read Only Memory (EPROM) parts. It also supports lower-cost one-time programmable (OTP) parts for both, small and large production runs. The very low-cost ROMed parts are supported for very large runs.

13. Low Power Consumption

For PIC16C6X and 16C7X, the operating voltage range is 3V-6V. The power consumption is very low. PIC16C6X takes less than 2 mA at 5 V and 4 MHz oscillator frequency and PIC16C7X takes 15 μ A at 3 V and 32 kHz oscillator frequency.

The Table 18.1 gives different features of PIC parts.

| PIC Feature | PIC16C81 | PIC16C66 | PIC16C71 | PIC16C74A |
|------------------------|----------------------------|----------------------------|-----------------|----------------------------|
| Program Memory X 14 | 1 K | 8 K | 1 K | 4 K |
| Data Memory X 8 | 36 Bytes | 368 Bytes | 36 Bytes | 192 Bytes |
| I/O Pins | 13 | 22 | 13 | 33 |
| Number of Instructions | 14-bit, 35 Instructions | 14-bit, 35 Instructions | 35 Instructions | 14-bit, 35 Instructions |
| Timer Modules | 1 + WDT | 3 + WDT | 1 + WDT | 3 + WDT |

| | | | | |
|--------------------------------|--------|--|----------------------------|--|
| Parallel Slave Port | – | – | – | Yes |
| Capture/Compare/PWM Module/(s) | 1 | 3 | – | 2 |
| Serial Communication | – | SPI ¹ /I ² C, USART | – | SPI ¹ /I ² C, USART |
| In-Circuit Serial Programming | Yes | Yes | Yes | Yes |
| Brown-out-Reset | – | Yes | – | Yes |
| Interrupt Sources | 3 | 10 | 4 | 12 |
| Maximum Speed | 20 MHz | 20 MHz | 20 MHz | 20 MHz |
| Additional Features | – | Power-on-reset | 4-channels of 8-bit ADC | 8-channels, 8-bit ADC, Power-on-reset |

Table 18.1 PIC parts with features

18.3 PIC16CXX

PIC16CXX are 8-bit microcontrollers. First, we will discuss different features of PIC16C6X microcontroller as an example of PIC16CXX family. Then we will study the details of PIC16C61 microcontroller.

18.3.1 Features of 16C6X Microcontroller

In this section we will discuss different features of PIC16C6X series.

18.3.1.1 Core Features

The core features of PIC16C6X microcontroller are stated below.

1. It is RISC processor and uses Harvard architecture.
2. The instruction set of PIC16C6X consists of only 35 single word instructions.
3. All instructions are single cycle instructions except for program branches which are two-cycle.
4. It has interrupt capability.
5. Its operating speed is,
DC - 20 MHz clock input,
DC - 200 ns instruction cycle
6. It has eight level deep hardware stack.
7. It supports direct, indirect and relative addressing modes.
8. It has inbuilt Power-on-Reset (POR).
9. It includes Power-up Timer (PWRT) and Oscillator Start-up Timer (OST).

10. It also contains Watchdog Timer (WDT) with its own on-chip RC oscillator for reliable operation.
11. It has programmable code protection.
12. It supports a power saving SLEEP mode.
13. It has selectable oscillator options.
14. It includes low power, high speed CMOS EPROM/ROM.
15. Its design is fully static.
16. It operates on wide voltage range : 2.5 V to 6.0 V.
17. It can be operated at commercial, industrial and extended temperature ranges.
18. Power consumption is low.
 - < 2 mA @ 5 V, 4 MHz
 - 15 μ A typical @ 3 V, 32 kHz
 - < 1 μ A typical standby current.

18.3.1.2 Peripheral Features

The peripheral features of PIC16C6X microcontroller are stated below.

1. It includes three timers.
 - a. **Timer 0 :** It is 8-bit timer/counter with 8-bit prescaler.
 - b. **Timer 1 :** It is 16-bit timer/counter with prescaler. It can be incremented during SLEEP mode via external crystal/clock.
 - c. **Timer 2 :** It is 8-bit timer/counter with 8-bit period register, prescaler and postscaler.
2. It includes Capture/Compare/PWM (CCP) module(s).
 - a. **Capture module :** It is 16-bit with max resolution 12.5 ns.
 - b. **Compare module :** It is 16-bit with max resolution 200 ns.
 - c. **PWM module :** Max. resolution is 10-bit.
3. It has Synchronous Serial Port (SSP) with SPI and I²C.
4. It contains Universal Synchronous Asynchronous Receiver Transmitter (USART/SCI).
5. It has Parallel Slave Port (PSP) which is 8-bits wide and it has external \overline{RD} , \overline{WR} and CS controls.
6. It contains Brown-out detection circuitry for Brown-out Reset (BOR).

18.3.2 Block Diagram

Fig. 18.2 shows the block diagram of PIC16C61.

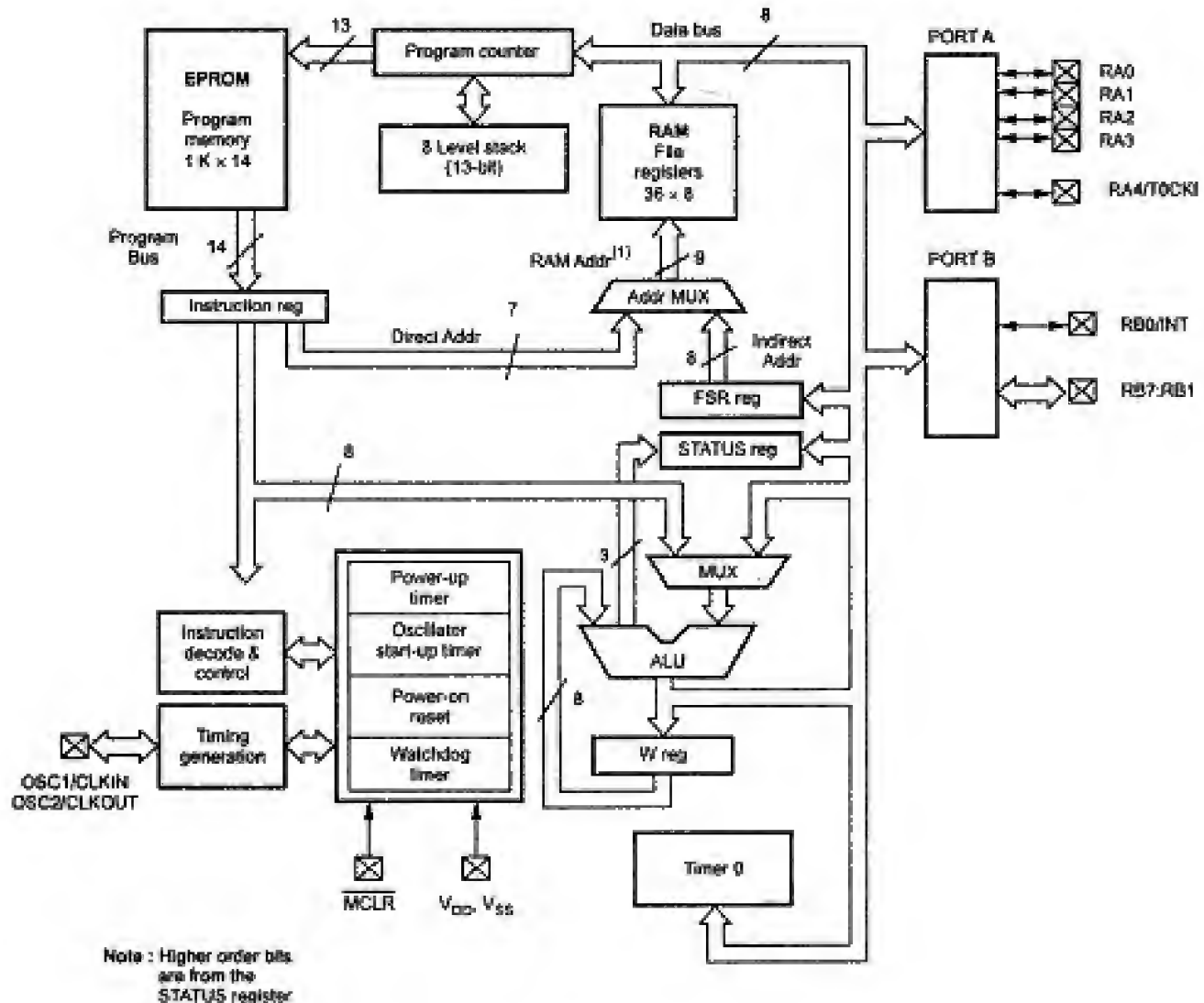


Fig. 18.2 PIC16C61 block diagram

18.3.2.1 Harvard Architecture

PIC16C61 uses a Harvard architecture in which program and data are accessed from separate memories using separate buses.

18.3.2.2 Address and Data Bus

Address bus in PIC16C61 is 14-bit wide. So it can access 1 K × 14 of program memory. All program memory is internal. Data bus in PIC16C61 is 8-bits wide.

18.3.2.3 ALU

The PIC16C61 contains 8-bit ALU. It is capable of performing arithmetic operations such as add, subtract, shift and logical operations. Unless otherwise mentioned, arithmetic operations are two's complement in nature. The inputs to the ALU are applied from two registers. In two-operand instructions, typically one operand is in the working register (W register) and the other operand is in a file register or an immediate constant. In single operand instructions, the operand is either in the W register or in a file register. The ALU may affect the values of bits in STATUS register (Carry bit, Digit carry bit, Zero bit) depending upon the instructions executed.

18.3.2.4 Registers

Table 18.2 gives the CPU registers in PIC16C61 with their widths.

| Sr. No. | CPU Register | Width |
|---------|---|-------------------|
| 1. | Working Register (W) | 8-bits |
| 2. | Status Register | 8-bits |
| 3. | File Selection Register (FSR) [Indirect Memory Address Pointer] | 8-bits |
| 4. | INDirect through FSR (INDF) | 8-bits |
| 5. | Program Counter Latch (PCLATH) | 5-bits |
| 6. | Program Counter Low Byte (PCL) | 8-bits |
| 7. | Program Counter Stack | 13-bits (8-level) |

Table 18.2 CPU registers

1. Working Register (W)

The working register, W is 8-bits wide. It is used for ALU operations. In two-operand instructions, typically one operand is in the working register (W). In single operand instructions, the operand is either in the W register or a file register. It is not an addressable register.

2. STATUS Register

The STATUS register contains the arithmetic status of the ALU, the RESET status and the bank select bits for data memory. The format of the status register is given below.

| R/W-0 | R/W-0 | R/W-0 | R-1 | R-1 | R/W-x | R/W-x | R/W-x |
|-------|-------|-------|-----------------|-----------------|-------|-------|-------|
| IRP | RP1 | RP0 | \overline{TO} | \overline{PD} | Z | DC | C |
| bit-7 | | | | | | | bit-0 |

R = Readable bit

W = Writable bit

- n = Value at POR reset

x = unknown

bit-7 : IRP : Register Bank Select bit (used for indirect addressing)

1 = Bank 2, 3 (100H - 1FFH)

0 = Bank 0, 1 (00H - FFH)

bit-6-5: RP1:RP0: Register Bank Select bits (used for direct addressing)

11 = Bank 3 (180H - 1FFH)

10 = Bank 2 (100H - 17FH)

01 = Bank 1 (80H - FFH)

00 = Bank 0 (00H - 7FH)

Each bank is 128 bytes.

bit-4: \overline{TO} : Time-out bit1 = After power-up, CLRWD \overline{T} instruction, or SLEEP instruction

0 = A WDT time-out occurred

bit-3: \overline{PD} : Power-down bit1 = After power-up or by the CLRWD \overline{T} instruction

0 = By execution of the SLEEP instruction

bit-2: Z: Zero bit

1 = The result of an arithmetic or logic operation is zero

0 = The result of an arithmetic or logic operation is not zero

bit-1: DC: Digit carry/borrow bit (for ADDWF, ADDLW, SUBLW, and SUBWF instructions) (For borrow the polarity is reversed).

1 = A carry-out from the 4th low order bit of the result occurred

0 = No carry-out from the 4th low order bit of the result

bit-0: C: Carry/borrow bit (for ADDWF, ADDLW, SUBLW, and SUBWF instructions) (For borrow the polarity is reversed).

1 = A carry-out from the most significant bit of the result occurred

0 = No carry-out from the most significant bit of the result

Note :

- a subtraction is executed by adding the two's complement of the second operand.
- For rotate (RRF, RLF) instructions, this bit is loaded with either the high or low order bit of the source register.

Fig. 18.3 Format of status register of PIC16C6X and PIC16C7X

The status register can be used as a destination register for any instruction like any other register. If the status register is used as a destination register for an instruction that affects the Z, DC or C bits, then the write to these three bits is disabled. These bits are set or cleared according to the device logic. Also, the \overline{TO} and \overline{PD} bits are only readable and not writable. So if the status register is used as a destination register for an instruction, then the result of an instruction may be different than intended. Therefore, for the

instructions (for example : BCF, BSF, SWAPF, MOVWF) which do not affect any status bits (Z, C or DC), the status register can be used for the storage purpose.

Note the following points while using status register.

1. For those devices that do not use 6th and 7th bits (RP1 and IRP), make these bits clear to ensure upward compatibility with future products.
2. The C and DC bits operate as a borrow and digit borrow bit respectively, in subtraction.

3. INDF (INDirect through FSR) and File Selection Register (FSR)

The INDF register is not a physical register. Addressing the INDF register will cause indirect addressing. Indirect addressing is possible by using the INDF register. Any instruction using the INDF register actually accesses the register pointed to by the File Select Register, FSR. Reading the INDF register itself indirectly (FSR = '0') will produce 00H. Writing to the INDF register indirectly results in a no-operation (although status bits may be affected). An effective 9-bit address is obtained by concatenating the 8-bit FSR register and the IRP bit (bit 7 of STATUS register), as shown in Fig. 18.4.

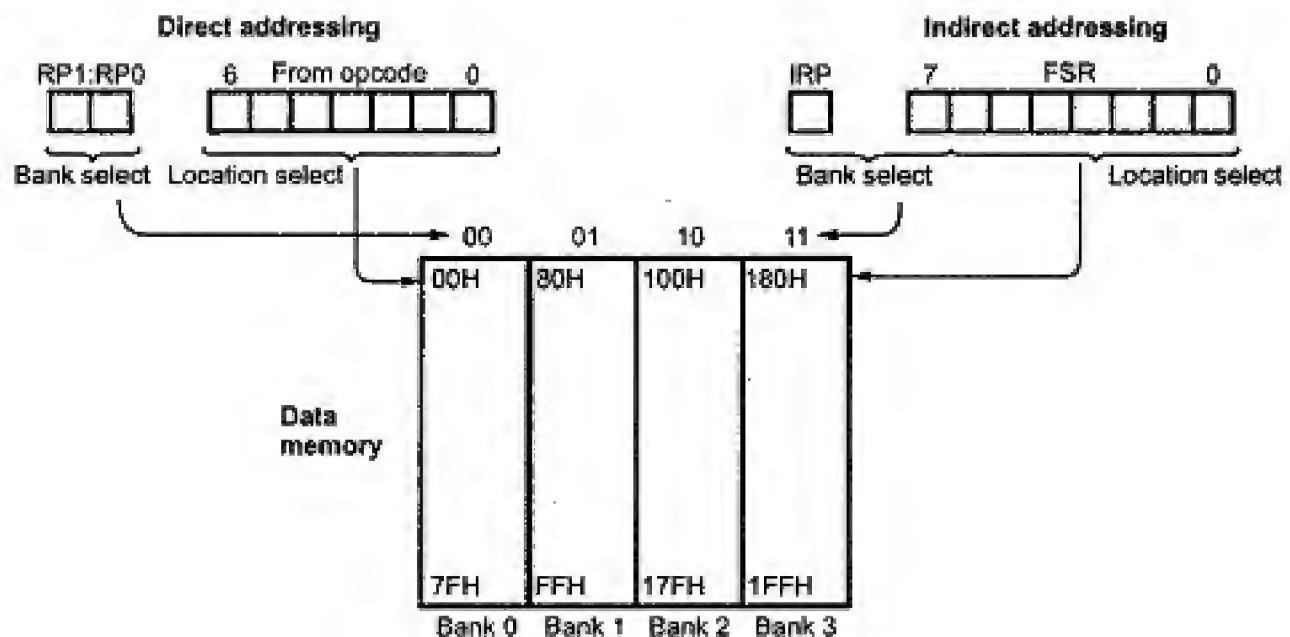


Fig. 18.4 Direct/indirect addressing

A simple program to clear RAM location 20H-2FH using indirect addressing is shown in the example given below.

EXAMPLE : INDIRECT ADDRESSING

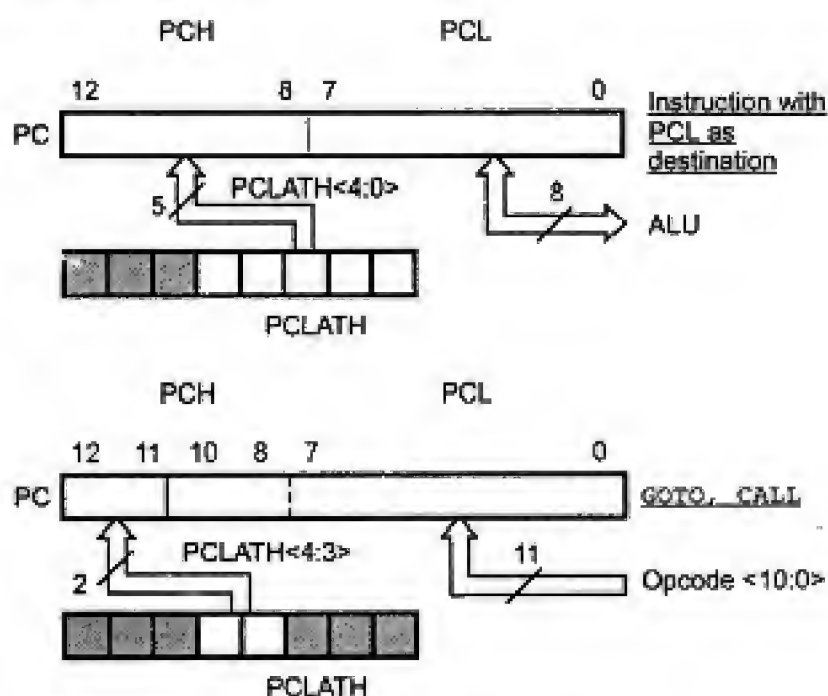
```

        movlw    0x20      ; initialize pointer
        movwf    FSR        ; to RAM
NEXT:   clrf     INDF       ; clear INDF register
        incf     FSR,F      ; inc pointer
        btfss    FSR,4      ; all done?
        goto     NEXT       ; NO, clear next
                          ; YES, continue

```

4. PCL and Program Counter Latch (PCLATH), Program Counter Stack

The program counter (PC) is 13-bits wide. The low byte comes from the PCL register, which is a readable and writable register. The upper bits (PC<12:8>) are not readable, but are indirectly writable through the PCLATH register. On any reset, the upper bits of the PC will be cleared. Fig. 18.5 shows the two situations for the loading of the PC. The upper example in the Fig. shows how the PC is loaded on a write to (PCLATH<4:0> → PCH). The lower example in the figure shows how the PC is loaded during a CALL or GOTO instruction (PCLATH<4:3> → PCH).

**Fig. 18.5 Loading of PC in different situations****Computed GOTO**

A computed GOTO is accomplished by adding an offset to the program counter (ADDWF PCL). When doing a table read using a computed GOTO method, care should be exercised if the table location crosses a PCL memory boundary (each 256 word block).

Stack

The PIC16CXX family has an 8 deep x 13-bit wide hardware stack. The stack space is not part of either program or data space and the stack pointer is not readable or writable. The PC is PUSHed onto the stack when a CALL instruction is executed or an interrupt causes a branch. The stack is POPed in the event of a RETURN, RETLW or a RETFIE instruction execution. PCLATH is not affected by a PUSH or a POP operation.

The stack operates as a circular buffer. This means that after the stack has been PUSHed eight times, the ninth push overwrites the value that was stored from the first push. The tenth push overwrites the second push (and so on). Thus, an 8-level stack will allow maximum 8 return addresses including return address from interrupt service subroutine.

- Note 1:** There are no status bits to indicate stack overflows or stack underflow conditions.
- Note 2:** There are no instructions mnemonics called PUSH or POP. These are actions that occur from the execution of the CALL, RETURN, RETLW, and RETFIE instructions, or the vectoring to an interrupt address.

18.3.2.5 Clocking Scheme/Instruction Cycle

The clock input (from OSC1) is internally divided by four to generate four non-overlapping quadrature clocks namely Q1, Q2, Q3, and Q4. Internally, the program counter (PC) is incremented every Q1, the instruction is fetched from the program memory and latched into the instruction register in Q4. The instruction is decoded and executed during the following Q1 through Q4. The clock and instruction execution flow is shown in Fig. 18.6.

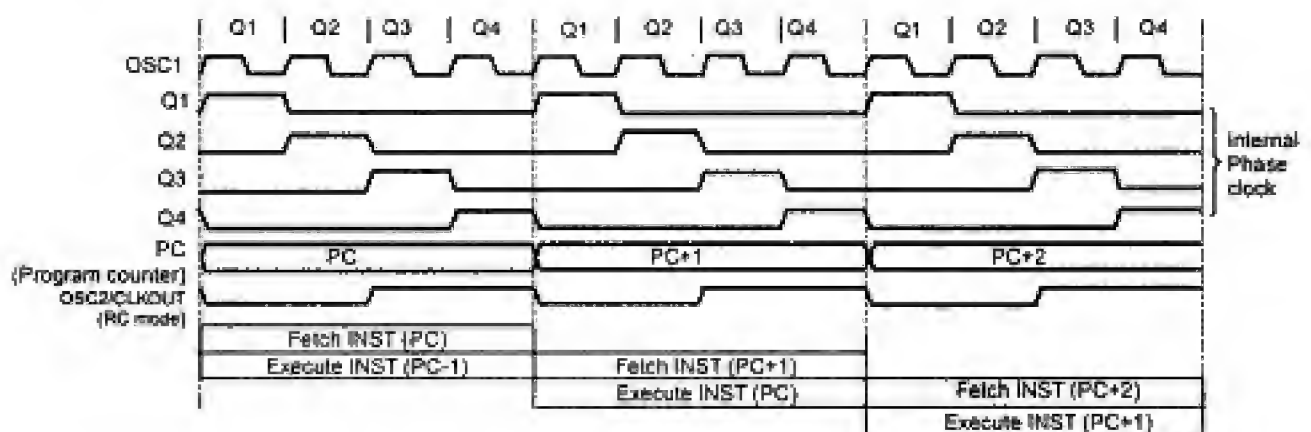
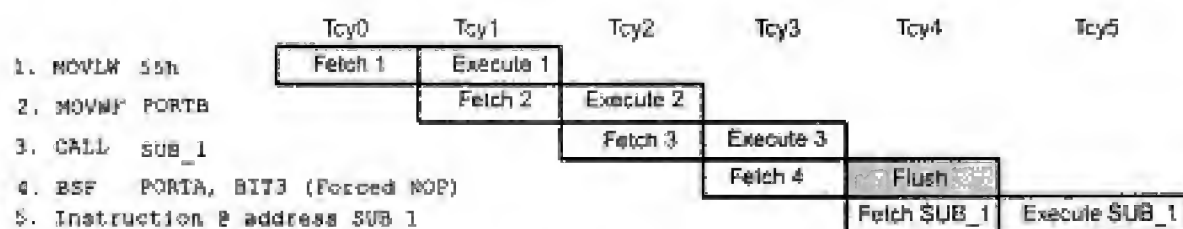


Fig. 18.6 Clock /Instruction cycle

Instruction Flow/Pipelining

An "Instruction Cycle" consists of four Q cycles (Q1, Q2, Q3, and Q4). The instruction fetch and execute are pipelined such that fetch takes one instruction cycle while decode and execute takes another instruction cycle. However, due to the pipelining, each instruction effectively executes in one cycle. If an instruction causes the program counter to change (e.g. GOTO) then two cycles are required to complete the instruction (Example 18.1).

➡ Example 18.1 : Instruction pipeline flow



All instructions are single cycle, except for any program branches. These take two cycles since the fetch instruction is "flushed" from the pipeline while the new instruction is being fetched and then executed.

A fetch cycle begins with the program counter (PC) incrementing in Q1.

In the execution cycle, the fetched instruction is latched into the "Instruction Register (IR)" in cycle Q1. This instruction is then decoded and executed during the Q2, Q3, and Q4 cycles. Data memory is read during Q2 (operand read) and written during Q4 (destination write).

18.3.3 Pin Diagram of PIC16C61

Fig. 18.7 shows the pin diagram of PIC16C61

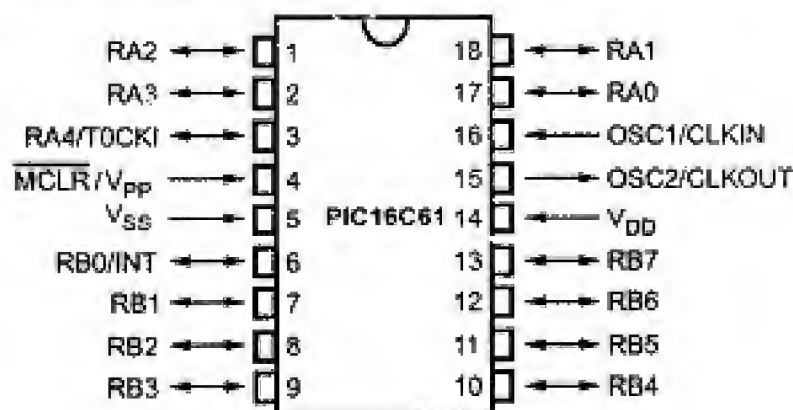


Fig. 18.7 PIC16C61

The following section describes the pins of PIC16C61.

| Pin Name | Pin Type | Description |
|--|----------|---|
| OSC1/CLKIN | I | Oscillator crystal input/external clock source input. |
| OSC2/CLKOUT | O | Oscillator crystal output. Connects to crystal or resonator in crystal oscillator mode. In RC mode, the pin outputs CLKOUT which has 1/4 the frequency of OSC1, and denotes the instruction cycle rate. |
| MCLR/V _{pp} | I/P | Master clear reset input or programming voltage input. This pin is an active low reset to the device. |
| RA0 RA1 RA2 RA3 RA4/T0CKI | I/O | PORTA is a bi-directional I/O port. RA4 can also be the clock input to the Timer0 timer/counter. Output is open drain type. |
| RB0/INT RB1 RB2 RB3 RB4 RB5 RB6 RB7 | I/O | PORTB is a bi-directional I/O port. PORTB can be software programmed for internal weak pull-up on all inputs. RB0 can also be the external interrupt pin. Interrupt on change pin. Interrupt on change pin. Interrupt on change pin. Serial programming clock. Interrupt on change pin. Serial programming data. |
| V _{SS} | P | Ground reference for logic and I/O pins. |
| V _{DD} | P | Positive supply for logic and I/O pins. |

Legend: I = Input O = Output I/O = Input/Output P = Power

Table 18.3 PIC16C61 Pinout description

18.3.4 Pin Diagram of PIC16C71

It is to be noted that PIC16C7X parts are almost same as PIC16C6X parts, with an exception that PIC16C7X includes an additional feature of an analog to digital converter.

Fig. 18.8 gives the pin diagram of PIC16C71.

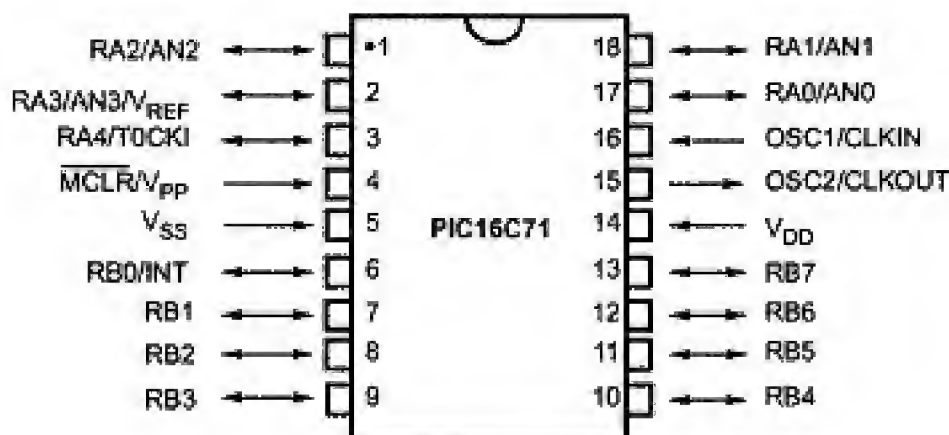


Fig. 18.8 PIC16C71

18.3.5 PIC Memory Organization

18.3.5.1 Program Memory

The PIC16C6X (and PIC16C7X) family has a 13-bit program counter. It can access 8 K × 14 program memory space. The device, PIC16C61 can address 1 K × 14 program memory having address range 0000H to 03FFH.

Fig. 18.9 shows PIC16C61 and PIC16C71 program memory map and stack.

A program counter is used to access 1 K × 14 program memory. The program counter is cleared after RESET. At location 0000H in the program memory, a 'Reset Vector', instruction is written which means 'goto mainline'. When an interrupt occurs, the address 0004H is automatically loaded into the program counter. At 0004H, 'goto interrupt service routine' instruction can be written. The interrupt service routine may be written somewhere else in the program memory space. Tables, main program and interrupt service routines can be written anywhere from 0005H to 03FFH.

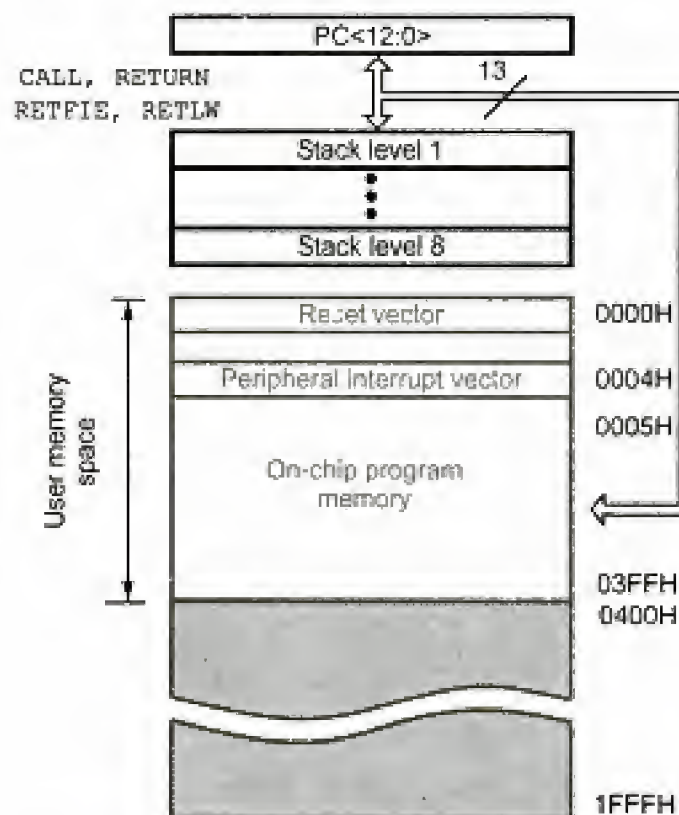


Fig. 18.9 PIC16C61 program memory map and stack

18.3.5.2 Data Memory

The data memory is partitioned into multiple banks which contain the General Purpose Registers and Special Function Registers. As seen earlier, bits RP1 and RP0 (6th and 5th bits) of status register are used to select the register banks as shown below.

| RP1 | RP0 | | |
|-----|-----|---|--------|
| 0 | 0 | → | Bank 0 |
| 0 | 1 | → | Bank 1 |
| 1 | 0 | → | Bank 2 |
| 1 | 1 | → | Bank 3 |

Each bank is 128 bytes.

Fig. 18.10 shows register file map of PIC16C61.

The lower locations of each bank are reserved for the Special Function Registers. Above the Special Function Registers are General Purpose Registers, implemented as static RAM. All implemented banks contain special function registers. Some "high use" special

| File address | | | File address |
|--------------|--------------------------------|------------------------------------|--------------|
| 00H | INDF ⁽¹⁾ | INDF ⁽¹⁾ | 80H |
| 01H | TMR0 | OPTION | 81H |
| 02H | PCL | PCL | 82H |
| 03H | STATUS | STATUS | 83H |
| 04H | FSR | FSR | 84H |
| 05H | PORTA | TRISA | 85H |
| 06H | PORTB | TRISB | 86H |
| 07H | | | 87H |
| 08H | | | 88H |
| 09H | | | 89H |
| 0AH | PCLATH | PCLATH | 8AH |
| 0BH | INTCON | INTCON | 8BH |
| 0CH | General Purpose Register | Mapped in Bank 0 ⁽²⁾ | 8CH |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| 2FH | | | AFH |
| 30H | | | B0H |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| 7FH | | | FFH |
| | Bank 0 | Bank 1 | |

■ Unimplemented data memory location; read as '0'.

Note 1: Not a physical register.

2: These locations are unimplemented in Bank 1. Any access to these locations will access the corresponding Bank 0 register.

Fig. 18.10 PIC16C61 register file map

function registers from one bank may be mirrored in another bank for code reduction and quicker access.

1. General Purpose Registers

These registers are accessed either directly or indirectly through the File Select Register (FSR).

For the PIC16C61, general purpose register locations 8CH-AFH of Bank 1 are not physically implemented. These locations are mapped into 0CH-2FH of Bank 0.

2. Special Function Registers

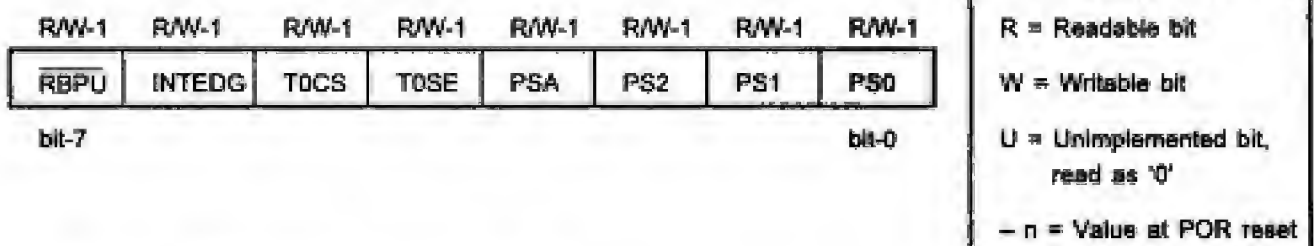
The Special Function Registers are registers used by the CPU and peripheral modules for controlling the desired operation of the device. These registers are implemented as static RAM.

The special function registers can be classified into two sets (core and peripheral).

3. OPTION Register

The OPTION register is a readable and writable register which contains various control bits to configure the TMR0/WDT prescaler, the external INT interrupt, TMR0, and the weak pull-ups on PORTB.

Fig. 18.11 shows the format of OPTION register.



bit-7: RBPU: PORTB Pull-up Enable bit

1 = PORTB pull-ups are disabled

0 = PORTB pull-ups are enabled by individual port latch values

bit-6: INTEDG: Interrupt Edge Select bit

1 = Interrupt on rising edge of RB0/INT pin

0 = Interrupt on falling edge of RB0/INT pin

bit-5: T0CS: TMR0 Clock Source Select bit

1 = Transition on RA4/T0CKI pin

0 = Internal instruction cycle clock (CLKOUT)

bit-4: T0SE: TMR0 Source Edge Select bit

1 = Increment on high-to-low transition on RA4/T0CKI pin

0 = Increment on low-to-high transition on RA4/T0CKI pin

- bit-3: **PSA:** Prescaler Assignment bit
 1 = Prescaler is assigned to the WDT
 0 = Prescaler is assigned to the Timer0 module
- bit-2-0: **PS2:PS0:** Prescaler Rate Select bits

| Bit Value | TMR0 Rate | WDT Rate |
|-----------|-----------|----------|
| 000 | 1 : 2 | 1 : 1 |
| 001 | 1 : 4 | 1 : 2 |
| 010 | 1 : 8 | 1 : 4 |
| 011 | 1 : 16 | 1 : 8 |
| 100 | 1 : 32 | 1 : 16 |
| 101 | 1 : 64 | 1 : 32 |
| 110 | 1 : 128 | 1 : 64 |
| 111 | 1 : 256 | 1 : 128 |

Note : To achieve a 1:1 prescaler assignment for TMR0 register, assign the prescaler to the Watchdog Timer.

Fig. 18.11 OPTION register (address 81h, 181h)

4. INTCON Register

The INTCON Register is a readable and writable register which contains the various enable and flag bits for the TMR0 register overflow, RB port change and external RB0/INT pin interrupts.

Fig. 18.12 shows the format of INTCON register.

| | | | | | | | |
|-------|-------|-------|-------|-------|-------|-------|-------|
| R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-x |
| GIE | PEIE | TOIE | INTE | RBIE | TOIF | INTF | RBIF |
| bit-7 | | | | | | | bit-0 |

R = Readable bit
 W = Writable bit
 U = Unimplemented bit, read as '0'
 - n = Value at POR reset
 x = unknown

- bit-7: **GIE:** ⁽¹⁾ Global Interrupt Enable bit
 1 = Enables all un-masked interrupts
 0 = Disables all interrupts
- bit-6: **PEIE:** ⁽²⁾ Peripheral Interrupt Enable bit
 1 = Enables all un-masked peripheral interrupts
 0 = Disables all peripheral interrupts

| | |
|--------|---|
| bit-5: | T0IE: TMR0 Overflow Interrupt Enable bit |
| | 1 = Enables the TMR0 overflow interrupt |
| | 0 = Disables the TMR0 overflow interrupt |
| bit-4: | INTE: RB0/INT External Interrupt Enable bit |
| | 1 = Enables the RB0/INT external interrupt |
| | 0 = Disables the RB0/INT external interrupt |
| bit-3: | RBIE: RB Port Change Interrupt Enable bit |
| | 1 = Enables the RB port change interrupt |
| | 0 = Disables the RB port change interrupt |
| bit-2: | T0IF: TMR0 Overflow Interrupt Flag bit |
| | 1 = TMR0 register overflowed (must be cleared in software) |
| | 0 = TMR0 register did not overflow |
| bit-1: | INTF: RB0/INT External Interrupt Flag bit |
| | 1 = The RB0/INT external interrupt occurred (must be cleared in software) |
| | 0 = The RB0/INT external interrupt did not occur |
| bit-0: | RBIF: RB Port Change Interrupt Flag bit |
| | 1 = At least one of the RB7:RB4 pins changed state to clear the interrupt |
| | 0 = None of the RB7:RB4 pins have changed state |

Fig. 18.12 INTCON register (address 0BH, 8BH, 10BH, 18BH)

18.3.6 Watchdog Timer

A watchdog timer (WDT) is a special timer with a specific function. It is usually used to prevent software crashes, i.e. endless loop. It works as follows : once started, the WDT increments an internal counter at the rate decided by free running on-chip oscillator. If the user program does not reset the counter, the counter overflows, which is used to reset the controller. In PIC microcontrollers, instruction CLRWDT is used to reset the counter. It sets the time-out bit (\overline{TO}) in the status register. This \overline{TO} bit is also set during the power up procedure . After time-out WDT timer resets \overline{TO} bit.

The normal time-out period of PIC watchdog timer is 18 ms. Therefore, if CLRWDT instruction is not executed within 18 ms, the counter overflows and it resets the PIC microcontroller. The time-out period for watchdog timer can be elongated by setting prescaler count. A prescaler with a division ratio of up to 1 : 128 can be assigned to the WDT under software control by writing to the OPTION register. Thus, time-out period up to 2.3 seconds can be realized.

The watchdog timer can be enabled or disabled by setting or resetting the bit 2 (WDTE) in the configuration word, respectively. In power-down mode, if the watchdog timer is enabled, it will be cleared but keeps running. Therefore, bit \overline{TO} in the status

register is set and bit \overline{PD} is reset. After timeout WDT can wake-up PIC from power down mode. Now it is interesting to see whether setting of \overline{PD} bit cause the power-up or time-out of WDT cause the power-up. The \overline{TO} and \overline{PD} bits in the status register can be used to determine the cause of power-up. If \overline{PD} is set to one power-up occurs due to setting of \overline{PD} bit. If \overline{TO} bit is zero, time-out has occurred and it is the cause of power-up.

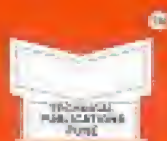
Review Questions

1. Give the features of PIC microcontroller.
2. Compare Harvard architecture and Von-Neumann architecture with neat diagrams.
3. What is brown-out-reset ?
4. Explain the term 'Watchdog Timer'.
5. Explain SLEEP mode of PIC.
6. Compare different features of PIC16C61 and PIC16C71.
7. Give the features of PIC16CXX.
8. Draw and explain the block diagram of PIC16C61.
9. Why is PIC program memory 14 bits wide ?
10. What do you mean by an 8-level program counter stack ?
11. Explain the stack operation in PIC16C61.
12. What do you mean by orthogonal instruction set ?
13. What is the role of INDF in indirect addressing mode ?
14. Write a short note on watchdog timer.

□□□

Contents

- Microprocessor and microcomputer system, Functional pin diagram and detailed architecture of 8085 microprocessor, Demultiplexing of address / data bus, Generation of control signals, Instruction set, Addressing modes, Programming for arithmetic and logical operation, Subroutine concepts.
- Functional pin diagram and architecture of 8031/51 microcontroller, Port structure, Instruction set and assembly language programming.
- Timer / counter, Modes of operation, Programming timer / counter, Interrupt structure and interrupts programming.
- Serial communication programming in 8051 (Only standard 8-bit UART mode).
- Memory interfacing (RAM, ROM, EPROM) - Basic concept in memory interfacing and address decoding.
- Programmable peripheral interface (8255) - Block diagram, Control words and modes and interfacing.
- Interfacing to external RAM and ROM, LED, Switch, 7-segment display, Multiplexed 7-segment display, Matrix key-board, Liquid crystal display, DAC, ADC, Stepper motor with programs.
- Buses and Protocols : RS 232, RS 485, , MODBUS, IEEE 488.
- Interfacing to EEPROM 93C46 / 56 / 66, 24C16 / 32 / 64, RTC DS1307.
- Conceptual study of various derivatives of 8051 microcontroller from different manufacturers like Atmel, Phillips etc. Introduction to PIC microcontroller.



Technical Publications Pune

1, Amit Residency, 412 Shaniwar Peth, Pune - 411030

☎(020) 24495496/97, Email : technical@vtubooks.com

Visit us at : www.vtubooks.com

First Edition : 2007-2008

Rs. 350/-

ISBN 978-81-8431-307-9



9 788184 313072